

Szymon Kołodziejczyk
Numer albumu: 091277

Silnik dla gier 2D z otwartym światem

Praca dyplomowa
na studiach I-go stopnia
na kierunku Informatyka

Promotor pracy dyplomowej:

dr inż. Grzegorz Łukawski
Katedra Systemów Informatycznych

Kielce 2024

POLITECHNIKA ŚWIĘTOKRZYSKA
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI

Studia stacjonarne
Kierunek: Informatyka

Rok akademicki: 2023/24

Zatwierdzam:
PRODZIEKAN
ds. Studenckich i Dydaktyki
Wydziału Elektrotechniki, Automatyki i Informatyki
.....
Prodziekan ds. studenckich i dydaktyki

ZADANIE NA PRACĘ DYPLOMOWĄ
STUDIÓW PIERWSZEGO STOPNIA

Wydano studentowi:

Szymon Kołodziejczyk

nr albumu: 91277

I. Temat pracy:

Silnik dla gier 2D z otwartym światem
2D open world game engine

II. Cel pracy:

Celem pracy jest projekt i implementacja silnika dla gier 2D z otwartym światem, z możliwością modyfikacji otoczenia (niszczenia, budowania) oraz z elementami fabularnymi (np. zadania, walka z wrogami, handel z postaciami niezależnymi). Proceduralne (pseudolosowe) generowanie elementów gry.

III. Plan pracy (zakres pracy):

- Przegląd technologii i zagadnień związanych z tematem pracy.
- Przegląd możliwości wybranych interfejsów i silników gier 2D/3D, uzasadnienie wyboru technologii do implementacji silnika.
- Projekt i implementacja silnika gier 2D.
- Projekt i implementacja przykładowej gry korzystającej z silnika.
- Testy działania, porównanie z innymi aplikacjami/grami o podobnej funkcjonalności.

IV. Uwagi dotyczące pracy: *brak uwag.*

V. Termin oddania pracy: *zgodnie z Regulaminem Studiów.*

VI. Konsultant: *Praca nie wymaga konsultanta.*

Opiekun merytoryczny

Dr hab. inż. Roman Deniziak, prof. PŚk

KIEROWNIK
Katedry Systemów Informatycznych
Wydziału Elektrotechniki, Automatyki i Informatyki
.....
Dr hab. inż. Roman Sienkiewicz Deniziak, prof. PŚk
(podpis)

Promotor pracy dyplomowej

Dr inż. Grzegorz Łukawski

.....
(podpis)

Temat pracy dyplomowej celem jej wykonania otrzymałem(am):

Kielce, dnia *30.10.2023*.....r.

.....
czytelny podpis studenta

Kielce, dnia 14.01.2024

Szymon Katońsiński 091207
Imię i nazwisko studenta, nr albumu
Szymon Katońsiński 26-212 Szymon
Adres zamieszkania
Studia pierwszego stopnia forma studiów stacjonarne
Studia pierwszego/drugiego stopnia, forma studiów stacjonarne/nie-stacjonarne
Informatyka specjalizacja grafika komputerowa
Kierunek, zakres
Dr. inż. Szymon Katońsiński
Promotor pracy dyplomowej

OŚWIADCZENIE

Przedkładając w roku akademickim 2023/24. promotorowi pracy dyplomowej studiów pierwszego/drugiego* stopnia, powołanemu przez Dziekana Wydziału Elektrotechniki, Automatyki i Informatyki Politechniki Świętokrzyskiej, pracę dyplomową pod tytułem: Systemy obrazu 2D z sterowanym ruchem

oświadczam, że:

- 1) przedstawiona praca dyplomowa została opracowana przeze mnie samodzielnie, stosownie do wskazówek merytorycznych opiekuna pracy,
- 2) przy wykonywaniu pracy dyplomowej wykorzystano materiały źródłowe, w granicach dozwolonego użytku wymieniając autora, tytuł pozycji i miejsce jej publikacji,
- 3) praca dyplomowa nie zawiera żadnych danych, informacji i materiałów, których publikacja nie jest prawnie dozwolona,
- 4) przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego/stopnia naukowego w wyższej uczelni,
- 5) niniejsza wersja pracy jest identyczna z treścią elektroniczną w systemie Archiwum Prac Dyplomowych.

Przyjmuję do wiadomości, że w przypadku ujawnienia w mojej pracy dyplomowej, stanowiącej podstawę nadania tytułu zawodowego, przypisania sobie przeze mnie autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego, rektor, w drodze decyzji administracyjnej, stwierdzi nieważność dyplomu.

Zostałem uprzedzony:

- 1) o odpowiedzialności kamej wynikającej z art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t. j. Dz. U. z 2022 r. poz. 2509 ze zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”,
- 2) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t. j. Dz. U. z 2022 r. poz. 574, ze zm.): „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”

Szymon Katońsiński
Czytelny podpis studenta

*niepotrzebne skreślić

Kielce, dnia ...14.01.2024...

.....
Imię i nazwisko studenta, nr albumu
.....
Adres zamieszkania
.....
Studia pierwszego/drugiego stopnia, forma studiów stacjonarne/niestacjonarne
.....
Kierunek, zakres
.....
Promotor pracy dyplomowej

OŚWIADCZENIE AUTORA PRACY

Zgodnie z ustawą z dnia 4 lutego 1994r. o prawie autorskim i prawach pokrewnych
(t.j. Dz. U. 2022 poz. 2509), wyrażam zgodę na udostępnianie mojej pracy dyplomowej dla celów naukowych
i dydaktycznych.

.....
Czytelny podpis studenta

Silnik dla gier 2D z otwartym światem

Streszczenie

Celem pracy jest projekt i implementacja silnika dla gier 2D z otwartym światem, z możliwością modyfikacji otoczenia (niszczenia, budowania) oraz z elementami fabularnymi (np. zadania, walka z wrogami, handel z postaciami niezależnymi). Proceduralne (pseudolosowe) generowanie elementów gry. Zostały wykonane testy wydajnościowe sprawdzające wykorzystane technologie. Zostały zaplanowane ścieżki dalszego rozwoju silnika

Słowa kluczowe: Gra 2D, symulacja terenu, silnik do gier, dynamiczne zmiany otoczenia.

2D open world game engine

Summary

The goal of the project is to design and implement a 2D game engine for an open-world environment, allowing for environment modification (destruction, construction), incorporating narrative elements (quests, combat with enemies, trading with non-playable characters). It will feature procedural (pseudo-random) generation of in-game elements. The performance tests assessing the utilized technologies have been conducted. Development paths for the engine's further advancement have been planned.

Keywords: 2D game, terrain simulation, game engine, Dynamic environment changes.

SPIS TREŚCI

1. WSTĘP	3
1.1 Motywacja pracy	3
1.2 Gry z otwartym światem	3
1.3 Opis elementów	4
1.3.1 Wielkość mapy	4
1.3.2 Ułatwienie przemieszczania się po świecie	4
1.3.3 Nieliniowe podejście w zwiedzaniu	5
1.4 Cel Pracy	5
1.5 Zakres Pracy	5
2. Projekt	7
2.1 Wymagania funkcjonalne	7
2.2 Wymagania нефункционалне	7
2.3 Mechaniki zaprojektowane w innych aplikacjach	7
2.3.1 Przechwytywanie obiektów na planszy	8
2.3.2 Przechowywanie danych odnośnie mapy i obiektów w plikach	9
2.3.3 Poruszanie przeciwników	10
3. Implementacja	11
3.1 Wybór technologii	11
3.1.1 Wybór języka	11
3.1.2 Wybór biblioteki graficznej	11
3.1.3 Wybór środowiska programowania	12
3.2 Implementacja Drzewa Czwórkowego	13
3.3 Implementacja Systemu zapisu	17
3.4 Implementacja Systemu szukania drogi	18
3.5 Generowanie terenu	20
3.5 Zaimplementowane narzędzia pomocnicze	25
3.6 Wzorce Projektowe	28
3.6.1 Singleton	29
3.6.2 Prototyp	30
4. Testy wydajnościowe	32
4.1 Test drzewa czwórkowego	32
4.2 Test zapisu na kilku plikach	34
4.3 Test wydajnościowy	35
5. Rozwój silnika	37
6. Podsumowanie	38
Bibliografia	39
Spis ilustracji	40
Spis Tabel	40

1. WSTĘP














Rozdział 1 wprowadza czytelnika w świat gier z otwartym światem, prezentując charakterystyczne elementy tego gatunku oraz przykładowe tytuły.

1.1 Motywacja pracy

Gry komputerowe to medium istniejące od kilkudziesięciu lat, co przyczyniło się do ogromnej popularności dostępu do tego środowiska. Dzięki postępowi technologii, w moim przypadku pojawiła się pasja do eksploracji gier, a następnie analizowania ich na czynniki pierwsze oraz próby własnoręcznego stworzenia odpowiednika danej produkcji. W ostatnich latach, inspirowany kilkoma doskonale wykonanymi grami z otwartym światem, zdecydowałem się na podjęcie wyzwania w tym konkretnym gatunku, co zaowocowało przedstawioną pracą inżynierską.

1.2 Gry z otwartym światem

Gry z otwartym światem cechują się udostępnieniem graczowi dużej przestrzeni, w której może swobodnie się poruszać. Ostatnio ten gatunek gier staje się coraz popularniejszy i częściej wykorzystywany przez twórców. W notowaniach platformy Steam[1] z okresu od 26 grudnia 2023 roku do 2 stycznia 2024 roku, wśród 12 najbardziej dochodowych produkcji, aż 6 gier zalicza się do kategorii gier z otwartym światem.

MIEJSCE		CENA	ZMIANA TYGODNIE	
1	 Counter-Strike 2	Free to Play	▲ 2	595
2	 Steam Deck	1 699,00 zł	▼ 1	97
3	 EA SPORTS FC™ 24	127,96 zł	▲ 1	25
4	 Baldur's Gate 3	-10% 249,90 zł 224,10 zł	▼ 2	41
5	 Cyberpunk 2077	99,50 zł		31
6	 Euro Truck Simulator 2	19,99 zł	▲ 2	509
7	 Lethal Company	45,99 zł	▼ 1	8
8	 War Thunder	Free to Play	▲ 3	109
9	 Forza Horizon 5	-50% 249,90 zł 124,95 zł	▼ 2	123
10	 Hogwarts Legacy	134,50 zł		2
11	 Red Dead Redemption 2	-67% 249,90 zł 82,46 zł	▼ 2	12
12	 ELDEN RING	-40% 249,90 zł 149,40 zł		2
13	 Grand Theft Auto V	-63% 172,80 zł 64,71 zł	▲ 2	468

Rysunek 1. Bestsellery w tygodniu 100 najlepszych produktów według przychodu z platformy Steam od wt., 26 grudnia 2023 do wt., 2 stycznia 2024

1.3 Opis elementów

Gry oparte na otwartym świecie różnią się znacząco od siebie w stylizowaniu świata możemy się wcielić w średniowieczną postać, przebyć dziki zachód lub zwiedzać świat w samochodzie. Mimo tych różnic gatunek ten posiada kilka wspólnych elementów.

1.3.1 Wielkość mapy

Gry z otwartym światem oferują graczom duże przestrzenie do eksploracji, umożliwiając im zwiedzanie rozległych i zróżnicowanych terenów. Istnieją dwa główne podejścia do tworzenia świata w tego typu grach.

Pierwszym z nich jest generowanie świata dynamicznie podczas rozgrywki gracza. Światy generowane w ten sposób powstają w uporządkowany sposób. Gry o takich światach posiadają olbrzymie mapy, które są tworzone na podstawie określonych wartości - tak zwanych "ziarnem". Przykładami popularnych gier, które generują świat w ten sposób, są: Minecraft, Necesse, Terraria.

Drugim podejściem, bardziej pracochłonnym, jest stworzenie świata przez ludzi, poprzez ręczne umieszczenie wszystkich elementów na mapie. Świat taki nie jest generowany losowo - przy każdym rozpoczęciu gry mapa pozostaje taka sama. Przykłady gier, które używają tego podejścia, to: GTA 5, Elden Ring, Cyberpunk 2077, The Legend of Zelda: Tears of The Kingdom[2].

1.3.2 Ułatwienie przemieszczania się po świecie

Gry te posiadają rozległe mapy, dlatego twórcy wprowadzili różne udogodnienia dla graczy, aby ułatwić im poruszanie się po świecie gry. Dodane rozwiązania mają na celu ułatwienie eksploracji, takie jak szybka podróż oraz dostęp do wierzchowców i pojazdów.

Szybka podróż jest zaprojektowana w taki sposób, aby gracze mogli łatwo powrócić do miejsc odwiedzonych wcześniej. W tym celu tworzone są obiekty umożliwiające teleportację, przyjmujące różne formy - od miejsc łaski w grze Elden Ring, po terminale w grze Cyberpunk 2077.

Wierzchowce[3], zazwyczaj konie, są powszechnie dostępne dla graczy, umożliwiając znacznie szybsze pokonywanie dużych odległości na mapie. Dodatkowo w niektórych grach istnieją pojazdy, takie jak samochody czy motocykle, które można nabyć w świecie gry. Czasami deweloperzy gier decydują się także na wprowadzenie możliwości budowy własnych pojazdów. Przykładem takiego podejścia jest gra "The Legend of Zelda: Tears of the Kingdom".

1.3.3 Nieliniowe podejście w zwiedzaniu

W grach z dużą otwartą mapą twórcy zazwyczaj pozostawiają graczom swobodę eksploracji świata, nie nakazując precyzyjnie, dokąd muszą się udać, aby przeżyć przygodę. Zamiast tego, subtelnie wskazują pewne miejsca, które mogą być warte odwiedzenia. Jako gracze, mamy możliwość skorzystania z tych wskazówek, ale również cieszymy się dużą swobodą, ponieważ nie ma sztywnych blokad ograniczających nasze poruszanie się.

Dlatego też istnieje możliwość eksploracji świata w sposób, który może różnić się od pierwotnych intencji twórców, umożliwiając graczom wyjątkowe i nieprzewidziane przygody. Brak wyraźnych ograniczeń pozwala na unikalne doświadczenia w grze, gdzie eksploracja może prowadzić do odkrywania nieoczekiwanych miejsc i wydarzeń.

1.4 Cel Pracy

Celem pracy jest zaprojektowanie silnika umożliwiającego tworzenie gier, które pozwolą na generowanie świata gry oraz rozbudowę tego świata poprzez dodanie pewnych elementów charakterystycznych dla gier typu "sandbox". Chcemy uwzględnić możliwość wpływania na teren, takie jak niszczenie bloków oraz budowanie, jak również implementację mechaniki walki z postaciami niezależnymi od gracza.

Silnik będzie oparty na zaawansowanych algorytmach generowania terenu, pozwalając graczom na eksplorację świata, który będzie unikalny przy każdym uruchomieniu gry. Dodatkowo planuje zaimplementować system interakcji z otoczeniem, który umożliwi niszczenie oraz konstruowanie elementów otaczającego świata. Mechanika walki z postaciami niezależnymi będzie stanowić istotny element rozgrywki, zapewniając graczom wyzwania i interakcję z otoczeniem.

Silnik będzie dążył do zapewnienia elastyczności i możliwości rozszerzania gry przez dodanie nowych funkcji czy elementów. Ma umożliwić tworzenie gier, które dadzą graczom pełną swobodę eksploracji, tworzenia oraz walki w otwartym i interaktywnym świecie.

1.5 Zakres Pracy

Zakres pracy obejmuje stworzenie gry na platformę Windows, która umożliwi graczom eksplorację różnorodnych terenów generowanych losowo w oparciu o "ziarno" wprowadzone przez gracza podczas tworzenia postaci.

Gra będzie zapewniać unikalne doświadczenia każdemu graczowi, pozwalając na generowanie losowych światów na podstawie określonego "ziarna" wprowadzonego podczas tworzenia postaci. Gracz będzie mógł wprowadzić swoje własne parametry, a na ich podstawie zostanie wygenerowany świat gry, który będzie różnił się od innych światów wygenerowanych przez inne "ziarna".

Celem gry będzie umożliwienie eksploracji zróżnicowanych terenów w wygenerowanym losowo świecie, zapewniając unikalne doświadczenia zależne od wybranego "ziarna". Gracz będzie miał możliwość odkrywania i badania różnorodnych miejsc, stworzonych na podstawie parametrów wprowadzonych na początku gry.

2. Projekt

Rozdział ten opisuje zbiór kluczowych funkcji i mechanik, są być zawarte w silniku gry, a następnie wykorzystane do stworzenia rozgrywki w grze z otwartym światem.

2.1 Wymagania funkcjonalne

Rozgrywka:

- Generowanie pseudolosowych map
- Interaktywna destrukcja obiektów na mapie.
- Edycja i dodawanie nowych obiektów do mapy.
- Walka z różnorodnymi przeciwnikami.
- Handel z postaciami niezależnymi.
- Tworzenie różnorodnych przedmiotów w grze.
- Używanie i aplikowanie różnego rodzaju przedmiotów.
- Funkcja zapisu i odczytu stanu gry.

Elementy poza rozgrywką:

- Możliwość dodawania nowych przedmiotów do gry z poziomu menu gry.
- Dodawanie nowych schematów receptur do gry z poziomu menu gry.

2.2 Wymagania нефunkcjonalne

- Gra dostępna pod systemy Windows 10, 11
- Odpowiednia optymalizacja by systemy posiadał niskie straty w klatkach na sekundę
- System powinien być prosty w dodawaniu nowych obiektów do rozgrywki
- Grafika powinna posiadać swój charakterystyczny element dla produkcji

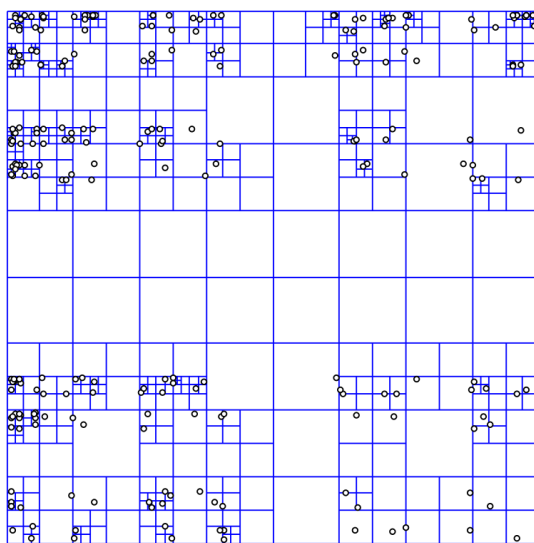
2.3 Mechaniki zaprojektowane w innych aplikacjach

W celu stworzenia pewnych mechanik w grze, korzystne jest przeanalizowanie funkcjonowania tych elementów w innych tytułach.

2.3.1 Przechwytywanie obiektów na planszy

Zbieranie obiektów na mapie może wydawać się proste, np. przechowywanie ich w strukturze danych typu lista lub wektor. Jednak w przypadku dużej liczby obiektów, operacja odnalezienia odpowiedniego może stać się kosztowna. Przy podziale mapy na fragmenty o rozmiarze np. 1600x900 pikseli, gdy obszar ten zawiera 5 obiektów, szukanie w pierwszej ćwiartce planszy nie jest zbyt kosztowne - wymaga sprawdzenia jedynie 5 obiektów w tym obszarze. Problem pojawia się, gdy liczba obiektów na tym obszarze wzrośnie z 5 do 50. W takim przypadku odnalezienie wszystkich obiektów w tej części planszy wymaga 50 porównań pozycji obiektów. W przypadku, gdy trzy obiekty próbują uzyskać dostęp do tych danych w jednej pętli, liczba porównań pozycji obiektów wzrasta do 150.

Istnieje jednak rozwiązanie tego problemu - można skorzystać ze struktury danych drzewa czwórkowego[4]. Ta struktura działa następująco: przechowuje wszystkie obiekty i w przypadku, gdy liczba obiektów osiągnie wartość maksymalną (podaną w kodzie programu), dzieli obszary na mniejsze regiony i grupuje obiekty w tych obszarach.



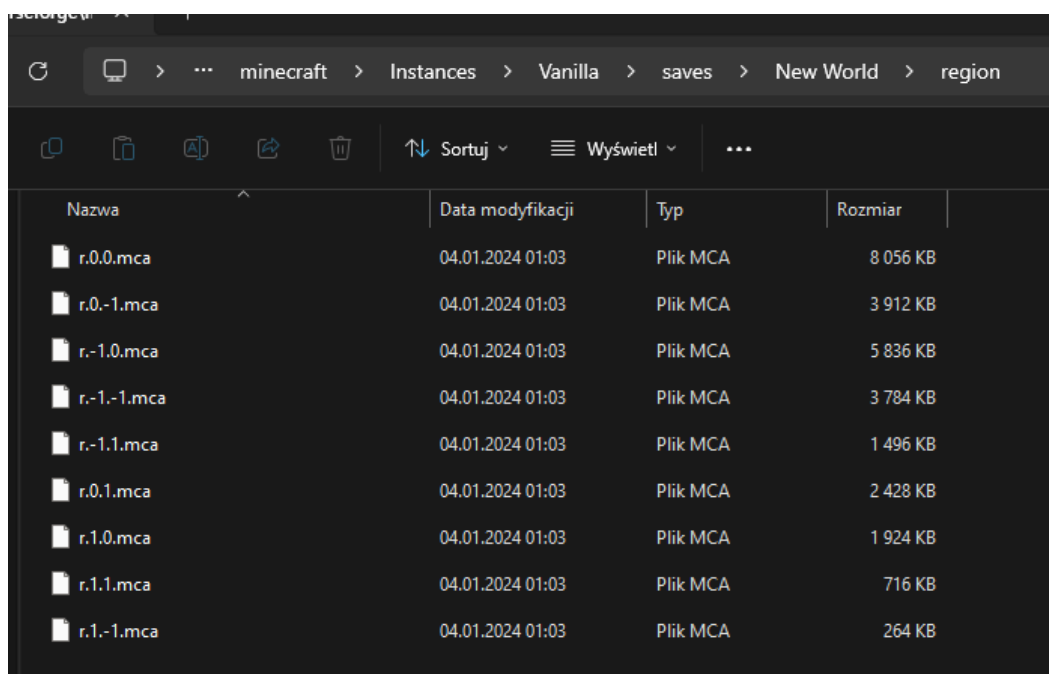
Rysunek 2. Drzewo czwórkowe wizualizacja struktury

Dzięki pogrupowaniu wszystkich obiektów na obszary z rysunku, możemy zauważyć, że aby uzyskać elementy z pierwszej ćwiartki, wystarczy dostarczyć do struktury drzewa obszar, z którego interesują nas obiekty. Następnie drzewo czwórkowe zwróci nam listę wszystkich poszukiwanych obiektów. Ta struktura działa na zasadzie sprawdzania obszarów: jeśli dany obszar nie odpowiada interesującemu nas obszarowi, pomija tę część drzewa. Proces wyszukiwania obiektów jest realizowany za pomocą funkcji rekurencyjnej.

2.3.2 Przechowywanie danych odnośnie mapy i obiektów w plikach

Przy zapisie wszystkich obiektów w jednym pliku pojawia się problem, który narasta wraz z dodawaniem nowych fragmentów mapy. Na początku, gdy plik zawiera tylko kilka fragmentów mapy, jego rozmiar nie jest znaczący. Kłopoty zaczynają się jednak, gdy generujemy setki kawałków terenu i próbujemy je zapisać w jednym pliku. Każde kolejne zapisywanie i wczytywanie terenu wymaga coraz więcej czasu, co drastycznie obniża wydajność i powoduje dyskomfort użytkownikowi.

Istnieje kilka potencjalnych rozwiązań tego problemu. Jednym z nich może być podział na mniejsze fragmenty, co pozwoliłoby zwiększyć wydajność, zwłaszcza przy obszarach o dużych rozmiarach. Podobny system plików został zastosowany w grze Minecraft, gdzie każdy plik zawierał kilkanaście fragmentów mapy.



Nazwa	Data modyfikacji	Typ	Rozmiar
r.0.0.mca	04.01.2024 01:03	Plik MCA	8 056 KB
r.0.-1.mca	04.01.2024 01:03	Plik MCA	3 912 KB
r.-1.0.mca	04.01.2024 01:03	Plik MCA	5 836 KB
r.-1.-1.mca	04.01.2024 01:03	Plik MCA	3 784 KB
r.-1.1.mca	04.01.2024 01:03	Plik MCA	1 496 KB
r.0.1.mca	04.01.2024 01:03	Plik MCA	2 428 KB
r.1.0.mca	04.01.2024 01:03	Plik MCA	1 924 KB
r.1.1.mca	04.01.2024 01:03	Plik MCA	716 KB
r.1.-1.mca	04.01.2024 01:03	Plik MCA	264 KB

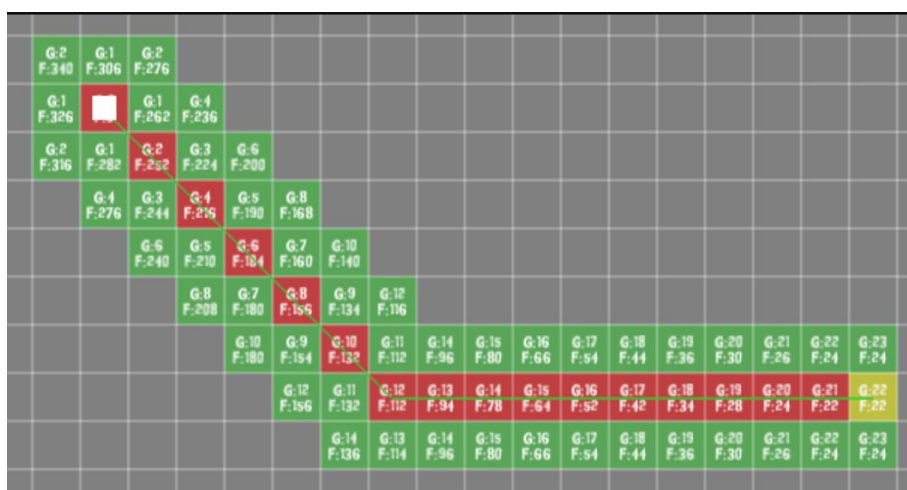
Rysunek 3. Rysunek przedstawia katalog z zapisanymi regionami odpowiednio ponumerowanymi.

W losowo generowanych prawie nieograniczonych mapach gry podejście do przechowywania danych jest ważne i zaniedbanie tego elementu może prowadzić do sporych problemów w przyszłości.

2.3.3 Poruszanie przeciwników

Tworzenie logicznych zachowań dla ruchu przeciwników może być wyzwaniem. Załóżmy, że chcemy stworzyć przeciwnika, który będzie podążał za graczem, utrzymując dystans od niego na poziomie 400 pikseli. Rozwiązanie wydaje się proste w implementacji, jednak pojawia się problem w sytuacji, gdy na linii prostej pomiędzy przeciwnikiem a graczem znajduje się przeszkoda, przez którą nie można przejść. W takim przypadku przeciwnik będzie próbował dostać się przez tę przeszkodę, dopóki gracz nie zmieni swojej pozycji. Takie zachowanie sprawia, że gra staje się zbyt łatwa, ponieważ przeciwnicy nie są w stanie dotrzeć do gracza, gdy ten znajduje się za odpowiednią przeszkodą.

Istnieje kilka algorytmów poszukujących ścieżek od punktu A do punktu B, z których jednym z popularnych jest algorytm szukania ścieżki A*[5]. Ten algorytm analizuje kolejne pola, sprawdzając ile kroków potrzebuje, aby dotrzeć do celu, oraz ile czasu zajęło mu dotarcie do danej pozycji.



Rysunek 4. Algorytm A* szukający ścieżki z białego kwadratu do żółtego punktu końcowego

Korzystanie z tego algorytmu pozwala przeciwnikom na bardziej inteligentne poruszanie się po mapie, co utrudnia ich zablokowanie. To stanowi większe wyzwanie dla graczy w aspekcie ucieczki oraz starć z nimi.

3. Implementacja

Rozdział 3 opowiada o procesie implementowania rozwiązań oraz wybór technologii w których postanie silnik.

3.1 Wybór technologii

Do stworzenia silnika do gier potrzebne jest wybranie środowiska w którym powstanie. Pośród języków programowania istnieje duża gamma możliwości.

3.1.1 Wybór języka

Ponieważ silniki do gier muszą zapewniać wysoką wydajność, języki interpretowane są wykluczone z listy preferowanych narzędzi do tworzenia rozbudowanych gier i silników gier. Zamiast tego preferowane są języki kompilowane, których kod jest przekształcany na język bardziej zbliżony do instrukcji komputerowych. Na rynku popularnymi językami są Java, C++[6], C#, C oraz Rust.

Porównując dostępne opcje, Java jest atrakcyjna, jeśli celem jest wydanie produktu na różne platformy. Jednakże ma pewne wady, takie jak automatyczne zwalnianie pamięci RAM w momencie, gdy obiekt nie jest dostępny w kodzie. To rozwiązanie, choć skuteczne, może prowadzić do niewłaściwego zarządzania pamięcią RAM, szczególnie gdy proste programy zajmują znacznie więcej pamięci, niż powinny, po jej zapelnieniu.

C, najstarszy język w zestawieniu, ma swoje ograniczenia wynikające z długiej historii. Brak obiektowości stwarza pewne problemy, wymaga poszukiwania alternatywnych rozwiązań. Jednak jego głównym atutem jest niski stopień abstrakcji, co czyni go najszybszym spośród wymienionych języków.

Rust to nowa, ciekawa opcja na rynku, mająca zaledwie 8 lat. Posiada kilka przydatnych mechanizmów, ale nauka tego języka może być wyzwaniem ze względu na odmienny charakter w porównaniu do tradycyjnych języków obiektowych.

Z dwóch języków, które są sukcesywnie wykorzystywane w grach i silnikach do gier, C++ jest szeroko stosowany w silniku Unreal Engine, podczas gdy C# jest wykorzystywany w silniku Unity. Analizując języki, w jakich napisane zostały te dwa popularne silniki do gier, naturalnym wyborem dla tworzenia własnego silnika wydaje się być język C++.

3.1.2 Wybór biblioteki graficznej

Po wyborze języka programowania, kluczowym krokiem jest wybór technologii do tworzenia okna gry oraz rysowania obiektów graficznych. Na rynku dostępne są różne biblioteki graficzne, a kilka z nich wyróżnia się szczególnie: SFML, OpenGL, Allegro oraz Raylib[7].

SFML to biblioteka, która oferuje prostą i szybką integrację dla wielu elementów multimedialnych, co czyni ją popularnym wyborem dla początkujących. Posiada intuicyjne API i obsługuje zarówno grafikę, dźwięk, jak i obsługę wejścia.

Allegro to również znana biblioteka graficzna, zapewniająca szeroki zakres narzędzi do tworzenia gier. Jest stabilna i ma długi staż, co przyciąga programistów poszukujących solidnej i sprawdzonej biblioteki.

OpenGL to bardziej nisko poziomowe API graficzne, oferujące zaawansowane możliwości renderowania grafiki 2D i 3D. Jest często wykorzystywane w profesjonalnych projektach do stworzenia zaawansowanych efektów wizualnych.

Raylib jest kolejną interesującą opcją. Choć mniej popularny niż SFML czy Allegro, jest on prosty w użyciu i oferuje szybkie narzędzia do tworzenia gier i aplikacji. Ma minimalistyczne API i wsparcie dla wielu platform.

Z dostępnych możliwości OpenGL jest najniższą opcją co komplikuje tworzenie silnika dlatego ta opcja została odrzucona z pozostałych 3 opcji najbardziej atrakcyjna jest biblioteka raylib mimo mniejszej popularności posiada dużo bardzo dobrych przykładów na stronie głównej z których zrozumienie pewnych elementów jest proste.

3.1.3 Wybór środowiska programowania

Na rynku istnieje kilka popularnych środowisk wspomagających programowanie w języku C++, oferujących czytelne interfejsy oraz możliwość dostosowania do indywidualnych potrzeb użytkowników.

CodeBlocks jest często wybierany przez początkujących programistów, ze względu na prostotę interfejsu i różnorodne narzędzia ułatwiające naukę programowania. Jego elastyczność i konfigurowalność pozwalają zarówno początkującym, jak i bardziej zaawansowanym programistom na wykorzystanie go w różnorodnych projektach.

Oprócz tego,

Visual Studio[8] jest potężnym narzędziem oferującym szeroki zakres funkcji dla programistów C++. Dzięki zaawansowanym funkcjom edytora kodu, debuggerowi, narzędziom do profilowania oraz integracji z różnymi platformami, jest często wybierane przez profesjonalistów i stosowane w projektach komercyjnych.

Natomiast CLion, skupia się na efektywnej pracy z językiem C++, oferując inteligentne wsparcie dla kodu, automatyczne uzupełnianie, analizę kodu oraz integrację z systemami kontroli wersji. Jest ceniony przez programistów za wydajność i przydatne narzędzia, które ułatwiają rozwój projektów. Jego interfejs jest bardziej wyspecjalizowany niż w przypadku CodeBlocks, co czyni go bardziej odpowiednim dla średnio zaawansowanych i zaawansowanych programistów.

Z podanych opcji najbardziej atrakcyjną opcją jest Visual Studio dzięki temu środowisku między innymi za pomocą pakietów nuget można w prosty sposób zainstalować bibliotekę która nas interesuje.

3.2 Implementacja Drzewa Czwórkowego.

Wspominając podpunkt 2.3.1 obiekty które mogą zmieniać pozycję lub obiekty potrzebujące blok pod nim będą przechowywane w strukturze drzewa czwórkowego.

Dla tego problemu została zaimplementowana klasa QuadTree jako że klasa opiera się na strukturze drzewa. Klasa posiada podwójny wskaźnik na obiekt QuadTree o nazwie tree. Spowodowane jest to faktem że po otwarciu kolejnego poziomu drzewa tworzymy tablicę 4 elementów które dzielą obszar na kolejne 4 ćwiartki. Podwójny wskaźnik jest także spowodowany brakiem konstruktora domyślnego gdyż musimy podać pozycję w której drzewo funkcjonuje

```
class QuadTree
{
    Rectangle pos;
    std::list<GameObject*> objects;
    QuadTree** tree = NULL;
};
```

Rysunek 5. Struktura drzewa w kodzie programu

Operacje na tej strukturze są rekurencyjne przez co nie mamy możliwości bezpośrednio odwołać się do danego kawałka drzew. Natomiast poprzez podanie odpowiednich fragmentów terenu z jakiego chcemy otrzymać obiekty struktura zwraca wszystkie obiekty odpowiadające danemu terytorium.

```

public:
    /// ...
    QuadTree(Rectangle pos);
    /// ...
    ~QuadTree();
    /// ...
    void addObj(GameObject* obj);
    /// ...
    void removeObj(GameObject* obj);

    void updatePos(GameObject* obj);
    /// ...
    bool hasObj(GameObject* obj);
    /// ...
    std::list<GameObject*> getObjectsAt(Rectangle pos);
    /// ...
    void draw();
};

```

Rysunek 6. Metody drzewa czwórkowego

Drzewo posiada kilka dostępnych metod, które są używane przez inne obiekty w systemie. Pierwszą z nich jest konstruktor, który tworzy obiekt, natomiast drugą jest destruktor, który odpowiada za zwalnianie pamięci całej struktury.

Przechodząc do często stosowanych metod po utworzeniu drzewa, istnieje metoda „addObj”, służąca do dodawania obiektu do odpowiedniej ćwiartki drzewa.

Z kolei metoda „removeObj” umożliwia usunięcie obiektu ze struktury i jest często używana przy całkowitym wyeliminowaniu obiektu ze sceny lub w przypadku, gdy obiekt przemieści się zbyt daleko od konkretnego fragmentu mapy.

„UpdatePos” jest kolejną metodą, wykorzystywaną do aktualizacji pozycji obiektu w strukturze drzewa. Zazwyczaj stosuje się ją dla obiektów poruszających się po mapie. W momencie zmiany pozycji obiektu, ta metoda zostaje wywołana dla danego obiektu.

„HasObj” to kolejna metoda, która sprawdza, czy dany obiekt istnieje w drzewie. Natomiast metoda „draw” służy do rysowania struktury, co jest przydatne do sprawdzania poprawności funkcjonowania drzewa.

Dzięki tym różnorodnym metodom drzewo jest w stanie zarządzać obiektami i ich interakcjami w sposób elastyczny i efektywny.

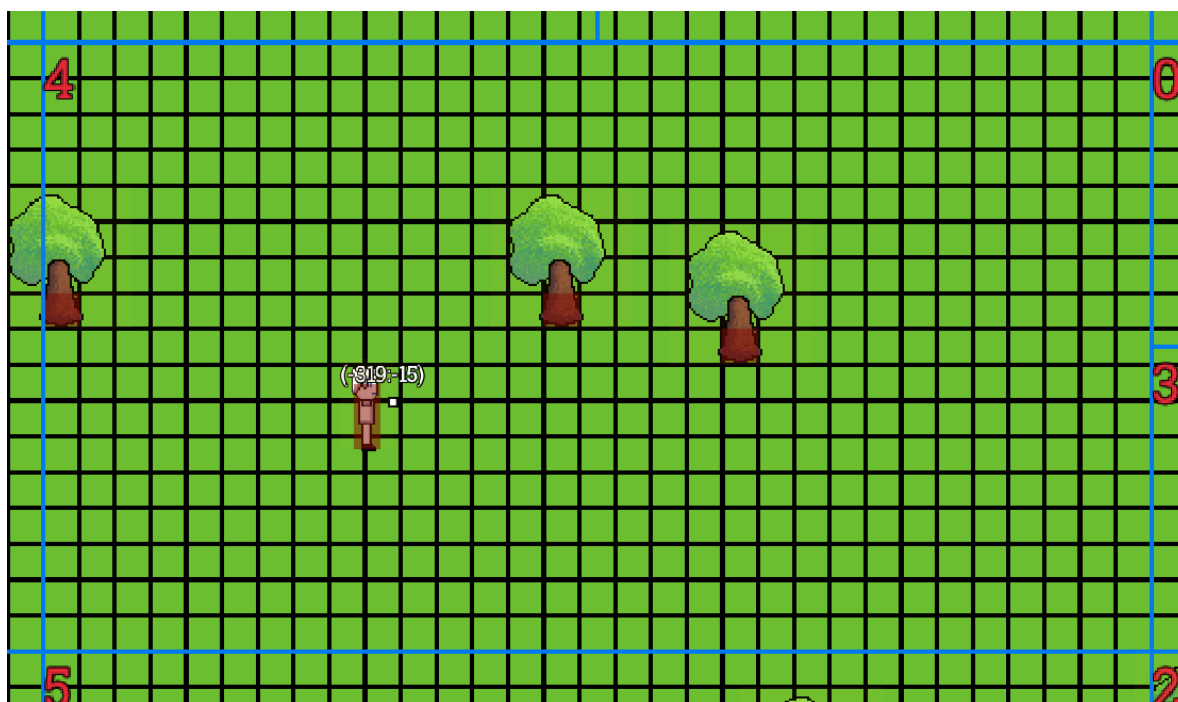

```

    /// ...
    void openTree();
    /// ...
    void closeTree();
public:

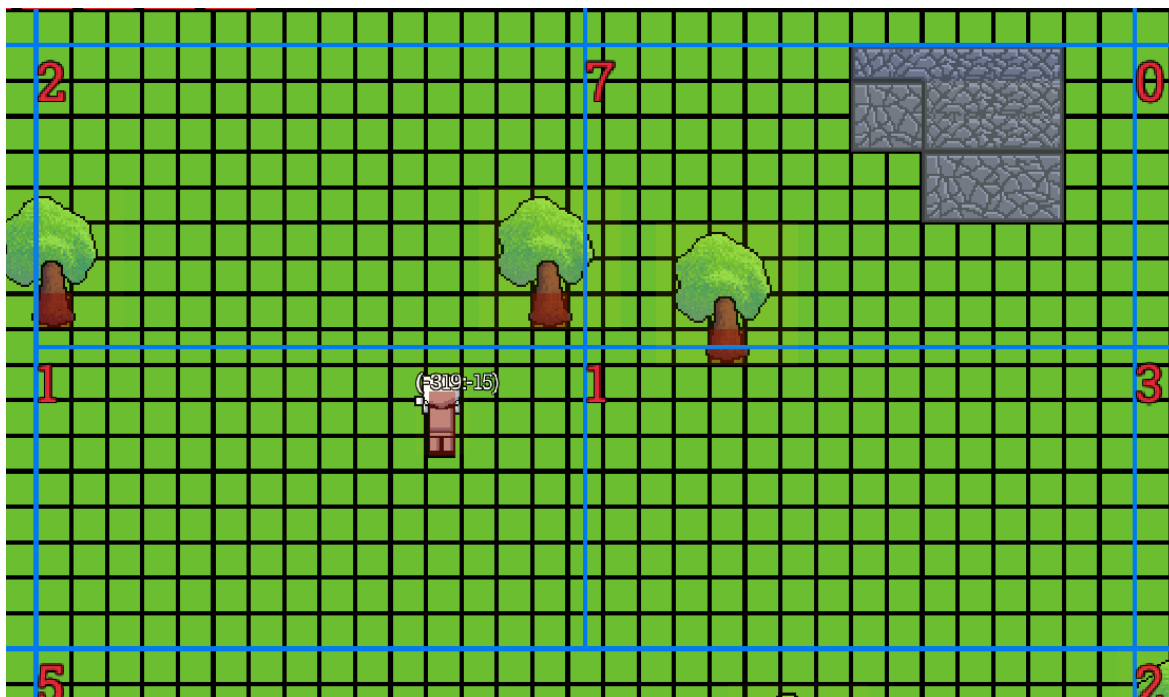
```

Rysunek 7. Prywatne metody klasy pozwalające zwiększyć drzewo oraz ucięcie fragmentu drzewa

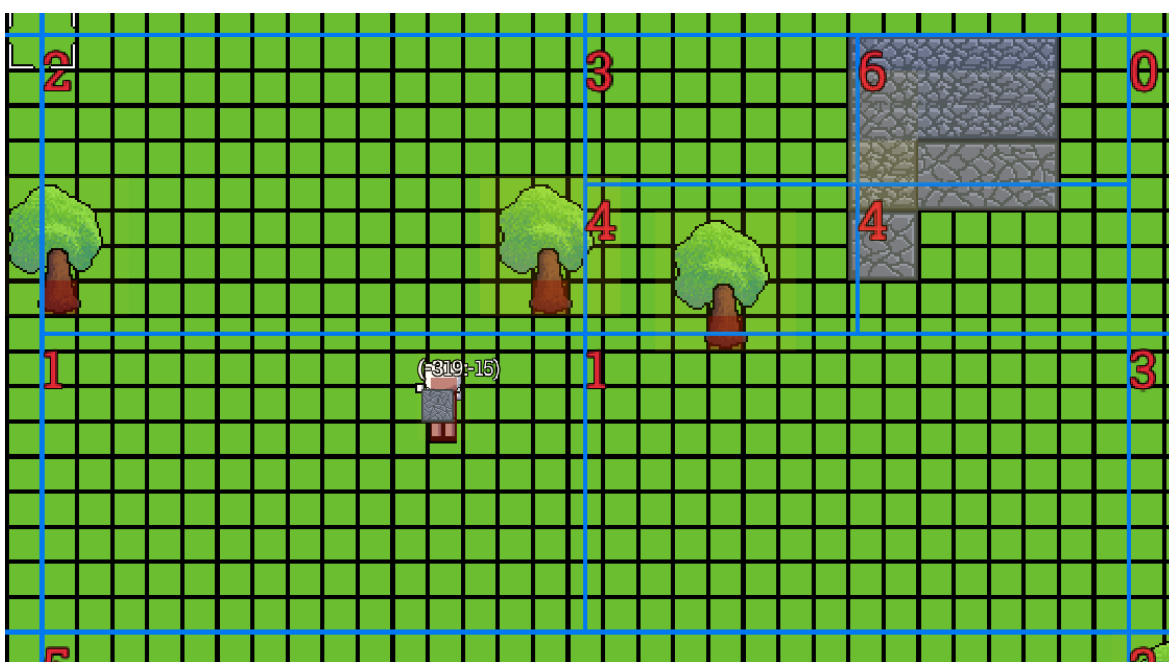
Istnieją jeszcze 2 metody prywatne które pomagają w rozszerzaniu struktury drzewa metoda „openTree” metoda ta dzieli obszar drzewa na 4 gdy w danym kawałku drzewa znajduje się więcej niż 8 obiektów. Natomiast metoda „closeTree” powoduje zamknięcie segmentu drzewa jeśli znajduje się tam mniej niż 4 obiekty by nie marnować pamięci na dużej liczby otwartych obszarów.



Rysunek 8. Przedstawienie struktury drzewa gdy w jego fragmencie znajduje się 8 obiektów.



Rysunek 9. Przedstawia strukturę drzewa po dodaniu dziewiątego obiektu



Rysunek 10. Przedstawienie struktury drzewa gdy zostanie dodane więcej obiektów w danej ćwiartce.

Z obrazków można zaobserwować, że z każdym kolejnym obiektem drzewo przechowujące obiekty się zwiększa się. Czerwony napis oznacza liczbę obiektów w ćwiartkach. Obiekty mogą wchodzić w kilka fragmentów drzewa, gdyż ich wielkość jest wystarczająco duża by dane się powtórzyły. Na rysunku 3.2.5 możemy zaobserwować, że drzewo z prawej górnej ćwiartki znajduje się też w dolnej dlatego wartość dolnej ćwiartki wynosi 1.

3.3 Implementacja Systemu zapisu

Zapisywanie danych z poszczególnych obiektów może stwarzać pewne wyzwania. Choć zapis obiektów w określonym, uporządkowanym formacie wydaje się rozsądny, dodawanie kolejnych zmiennych do obiektów może spowodować trudności w odczycie danych.

Obecnie powszechnie stosowanymi formatami plików do tego celu są JSON[9] oraz XML. JSON jest elastycznym sposobem reprezentacji danych, dlatego też został wybrany do wykorzystania w projekcie. Posiada on także kluczową zaletę: jako że zapisuje dane w formie klucz-wartość, umożliwia nam sprawdzenie istnienia określonego klucza. Jest to istotne, ponieważ po wprowadzaniu zmian w kodzie lub dodawaniu nowych funkcji, pozwoli nam odczytać świat, który został wygenerowany w poprzednich wersjach aplikacji. Dzięki temu zachowamy spójność danych nawet w przypadku modyfikacji i ewolucji projektu.

```
void GameObject::saveToJson(nlohmann::json &j)
{
    j["Pos"][0] = (int)pos.x;
    j["Pos"][1] = (int)pos.y;
    j["Pos"][2] = (int)pos.width;
    j["Pos"][3] = (int)pos.height;
    j["Name"] = name;
    j["ID"] = ID;
    j["ObjType"] = getType();
    j["ObjClass"] = getType();
}
```

Rysunek 11. Zapisywanie danych abstrakcyjnej klasy GameObject

```
void GameObject::readFromJson(nlohmann::json& j)
{
    pos.x = j["Pos"][0];
    pos.y = j["Pos"][1];
    pos.width = j["Pos"][2];
    pos.height = j["Pos"][3];
    name = j["Name"];
    ID = j["ID"];
}
```

Rysunek 12. Wczytywanie danych abstrakcyjnej klasy GameObject

W pierwotnej klasie, z której wywodzą się wszystkie obiekty w grze istnieje wirtualna metoda do zapisywania i wczytywania danych do pliku Json. W kolejnych klasach dziedziczących dodaje się kolejne dane do zapisu i odczytu jednocześnie wywołujące funkcję pierwotną

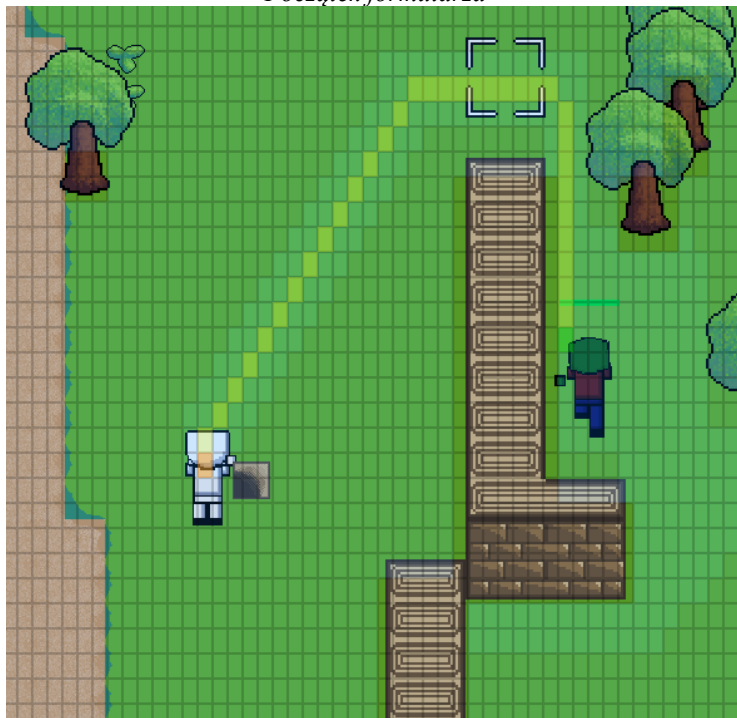
3.4 Implementacja Systemu szukania drogi

Algorytmy wyszukiwania optymalnej ścieżki są powszechnie stosowane do tworzenia inteligencji poruszających się niezależnych postaci w grach. Istnieje wiele sposobów implementacji algorytmów wyszukiwania optymalnej ścieżki, jednak najczęściej używanym rozwiązaniem jest algorytm A*.

Algorytm A* opiera się na przemieszczaniu się przez kolejne połączenia punktów, zapisując czas potrzebny na dotarcie do danego punktu oraz odległość od celu, do którego dążymy. Jego złożoność nie jest zbyt skomplikowana, jednak pojawiają się problemy z jego wykorzystaniem w realnym scenariuszu.

Choć algorytm ten jest stosunkowo klarowny i wydajny, jego implementacja w praktycznych warunkach czasami napotyka na trudności związane z niestandardowymi scenariuszami i specyficznymi warunkami środowiska w grze.

Początek formularza



Rysunek 13. Przeciwnik wyszukuje drogi do gracza.

Zgodnie z przedstawieniem na rysunku, przeciwnik ma wyznaczoną trasę, którą podąża, aby dotrzeć do gracza. Żółta ścieżka wyznacza bezpośrednią drogę do czerwonego prostokąta, symbolizującego pozycję gracza. Otaczająca jaśniejsza ścieżka ilustruje, które elementy drogi zostały wzięte pod uwagę podczas wyznaczania trasy przeciwnika.

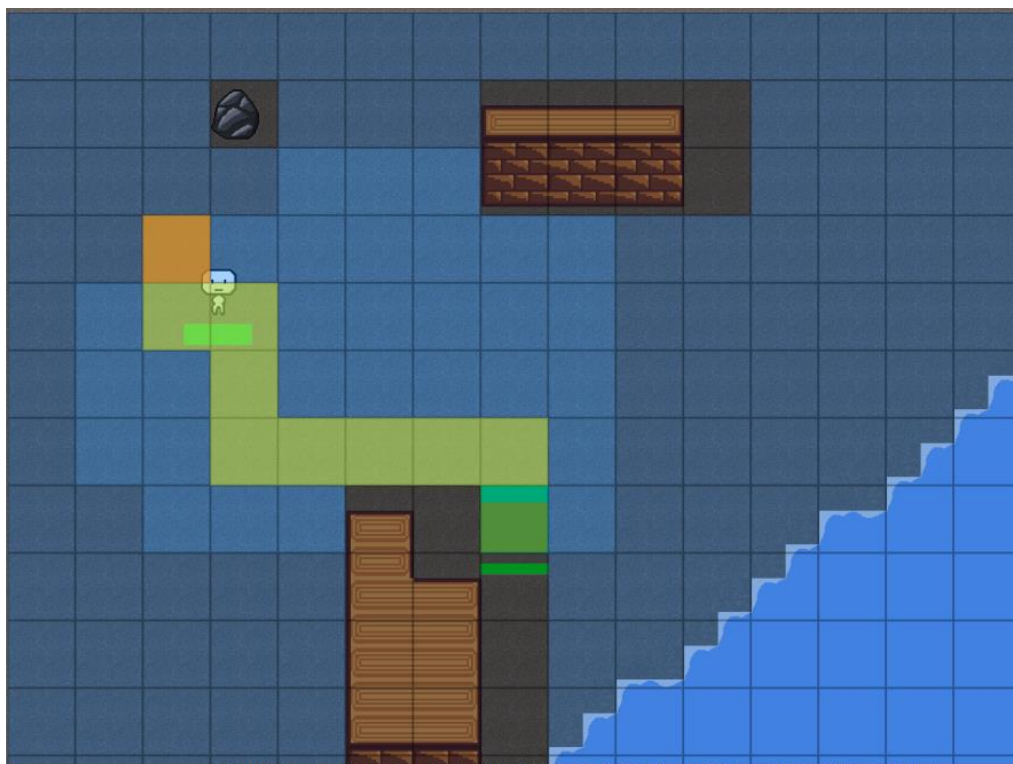
W obszarze przy ścianie widoczne są bardziej wyblakłe oznaczenia, reprezentujące obszar, przez który przeciwnik nie jest w stanie przejść. To wyróżnienie wskazuje na nieprzejezdny teren dla przeciwnika, stanowiący przeszkodę na jego drodze do gracza.



Rysunek 14. Przeciwnik przeszedł przez wybudowaną przeszkodę

Po kilku sekundach obserwacji można zauważyć płynność ruchu przeciwnika bez żadnych trudności. To zoptymalizowane rozwiązanie, gotowe już w pierwszej wersji. W pierwotnej wersji, prostokąty reprezentujące przeciwnika były większe, co powodowało problemy z wyznaczaniem poprawnej ścieżki w przypadku przeszkód. Były to zaznaczone pola, które wprowadzały błędy w ścieżce poruszania się przeciwnika.

Jednakże wyznaczanie ścieżki po każdym pojedynczym pikselu nie jest rozsądnym rozwiązaniem. Obliczenia i przechowywanie tak szczegółowej ścieżki w pamięci są zbyt kosztowne w stosunku do uzyskanego rezultatu. Dlatego podział postaci na elementy o wysokości i szerokości podzielonej na cztery części jest odpowiednim rozwiązaniem. Ten sposób działania działa właściwie i zapewnia optymalne rezultaty.



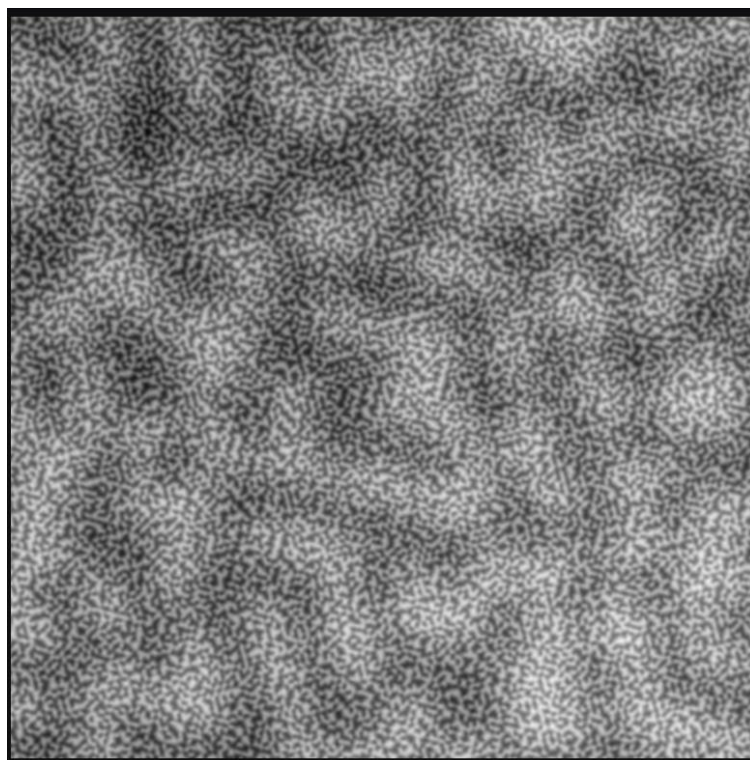
Rysunek 15. Pierwszy system szukania ścieżki

Z pierwotnej wersji wynika, że algorytm miał trudności z dokładnym określeniem obszarów, przez które nie można się poruszać. Zauważalne jest, że po lewej stronie od miejsca rozpoczęcia wyszukiwania trasy, istnieje możliwość przejścia w lewo. Jednakże, obszar ten jest zaznaczony jako wygaszony, co uniemożliwia dostęp do tego pola.

3.5 Generowanie terenu

Aby stworzyć świat generowany losowo z uporządkowaną strukturą, wykorzystuje się algorytmy generujące tzw. szum. Często używany jest szum Perlin, ale dla uzyskania interesującego terenu stosuje się również szum OpenSimplex2 ValueCubic. Biblioteka do generowania szumów jest dostępna na GitHub[10].

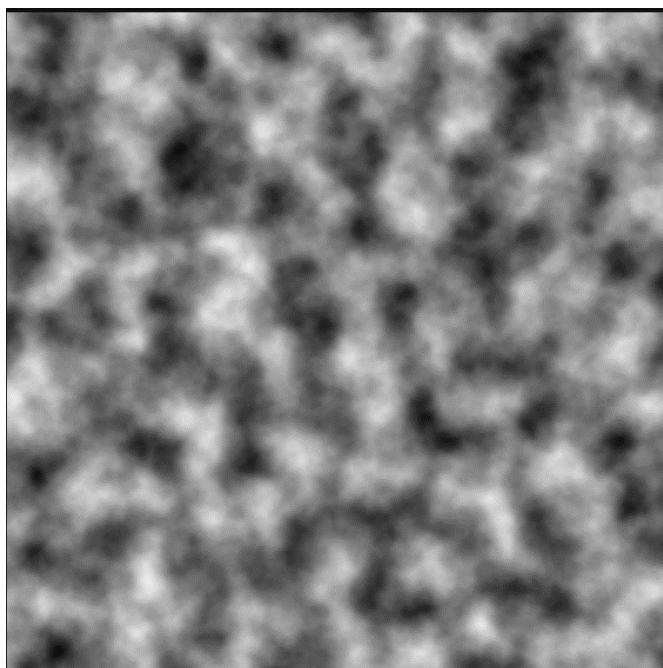
Korzystając z istniejącego rozwiązania do generowania szumów, konieczne jest właściwe dostosowanie parametrów szumów. Po ich modyfikacji uzyskano następujące rezultaty: Pierwszy szum został wygenerowany w celu określenia wartości terenu, który będzie determinował umiejscowienie obiektów na mapie, takich jak drzewa, kamienie czy inne rodzaje konstrukcji. Drugi szum określa obszary terenu oraz lokalizacje zbiorników wodnych.



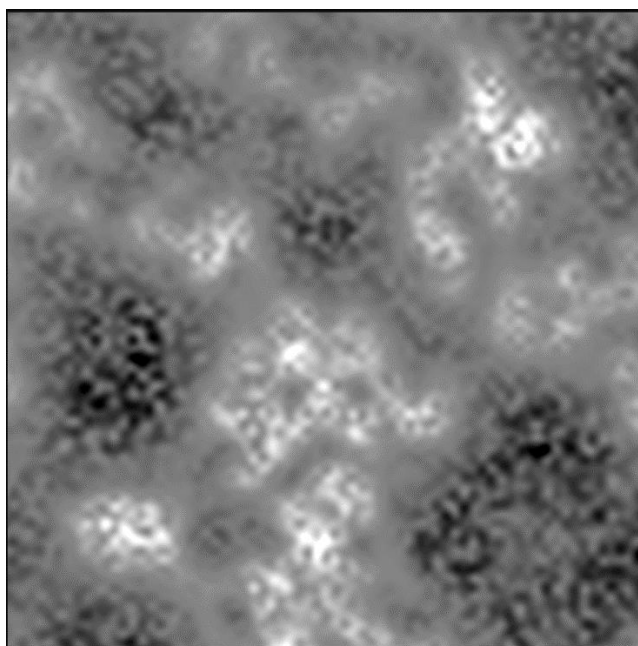
Rysunek 16. Szum odpowiedzialny za konstrukcje na mapie.



Rysunek 17. Szum odpowiedzialny za określenie występowania wody.



Rysunek 18. Szum odpowiedzialny za określenie przyjazności terenu.



Rysunek 19. Szum odpowiedzialny za określenie temperatury występującej na danym obszarze

Trzeci szum odpowiada za uwarunkowania terenu, na którym się przemieszczamy. W przyszłości możliwe jest wykorzystanie tej wartości do regulacji liczby generowanych przeciwników, bądź też określenia obszarów, gdzie mogą pojawić się przyjazne istoty, takie jak zwierzęta lub niezależne postacie zajmujące się handlem.

Ostatni, czyli czwarty, szum zostanie wykorzystany do określenia temperatury obszarów. Dzięki temu będziemy w stanie zlokalizować na mapie obszary, gdzie mogą występować różne typy klimatyczne, takie jak pustynie, krainy ośnieżone czy zwykłe lasy.

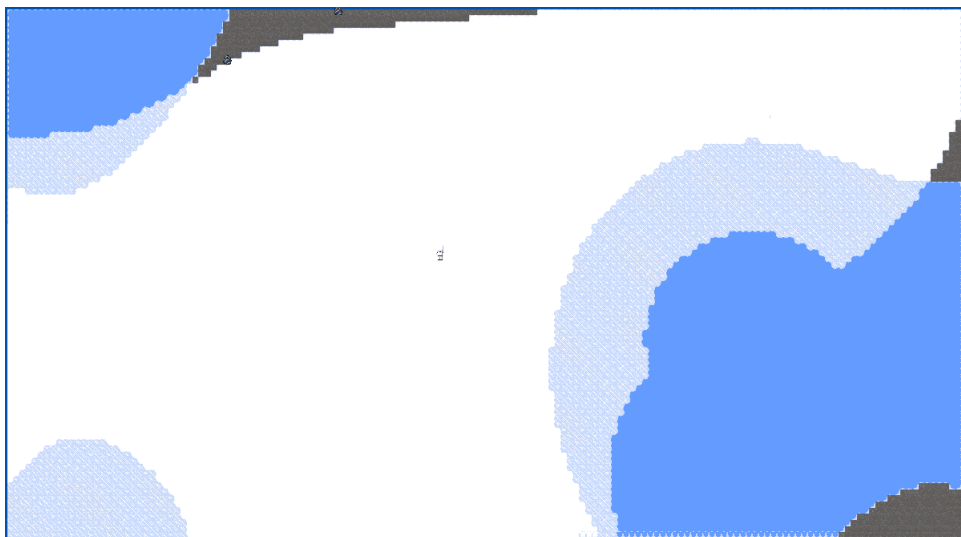

```

if ((friendlyV < -0.3 && temperatureV > 0.6) || (friendlyV < -0.1 && temperatureV > 0.8))
{
    generateDesertBiom(factory, pos, waterV, terrainV, x, y);
}
else if ((friendlyV < -0.3 && temperatureV < -0.4) || (friendlyV < -0.1 && temperatureV < -0.6))
{
    generateSnowBiom(factory, pos, waterV, terrainV, x, y);
}
else if ((friendlyV > 0.0 && temperatureV > -0.3f) || friendlyV > 0.8f)
{
    generateForestBiom(factory, pos, waterV, terrainV, x, y);
}
else
{
    generateStoneBiom(factory, pos, waterV, terrainV, x, y);
}

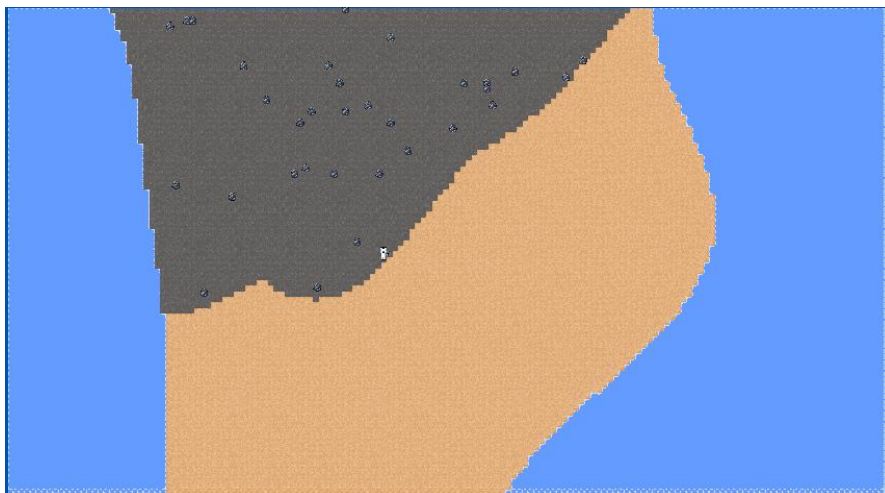
```

Rysunek 20. Fragment kodu determinujący teren jaki zostanie wygenerowany

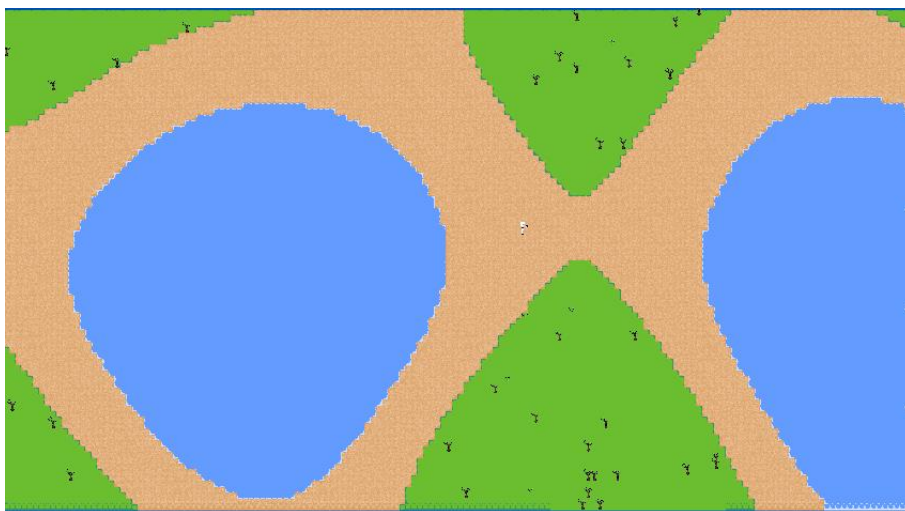
W kodzie, jak można zaobserwować na przedstawionym powyżej obrazku (Rysunek 3.4.5), wybór generowanego typu terenu zależy głównie od dwóch czynników: przyjazności terenu oraz temperatury. Przy niskiej przyjazności terenu mogą zostać wygenerowane biomy piaskowe lub zimowe, zależnie jednak od temperatury, która odgrywa kluczową rolę w procesie generowania biomów. Im niższa temperatura, tym większe prawdopodobieństwo wygenerowania terenu zimowego, a im wyższa temperatura, tym większe prawdopodobieństwo wystąpienia pustyni. Należy również zaznaczyć, że wartości mieszczą się w przedziale od -1 do 1, gdzie -1 oznacza na przykład bardzo zimne miejsce, a 1 bardzo gorące. Jeśli teren jest przyjazny, zostanie wygenerowany biom leśny, natomiast jeśli żaden z powyższych warunków nie będzie spełniony, generowany będzie biom skalny. Efekty tego rozwiązania generuje teren taki jak przedstawiony poniżej. Sposób generowania terenu został. Sposób generowanie terenu został obszerniej wytłumaczony w wykładzie znajdującym się na platformie Youtube [11].



Rysunek 21. Wygenerowany teren zimowy



Rysunek 22. Wygenerowana pustynia



Rysunek 23. Wygenerowany Las z plażą

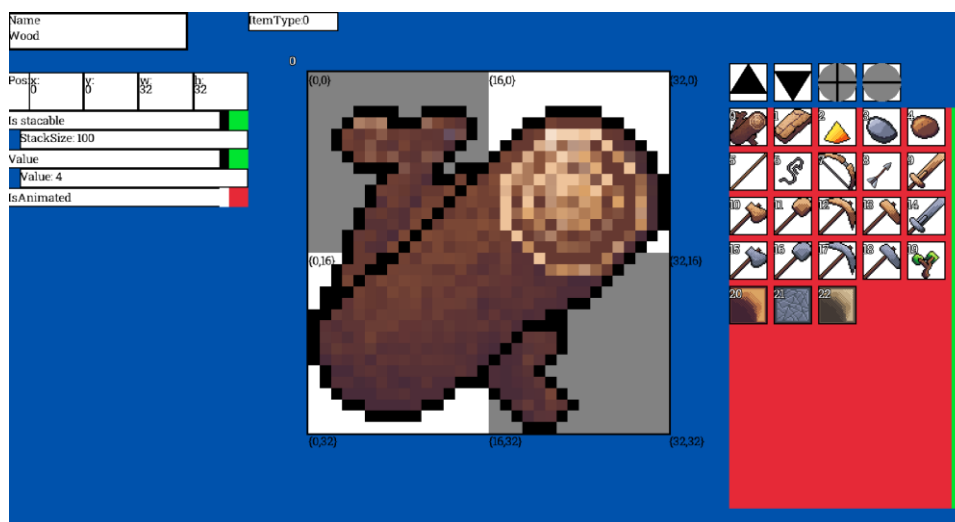


Rysunek 24. Przykładowa wygenerowana wyspa

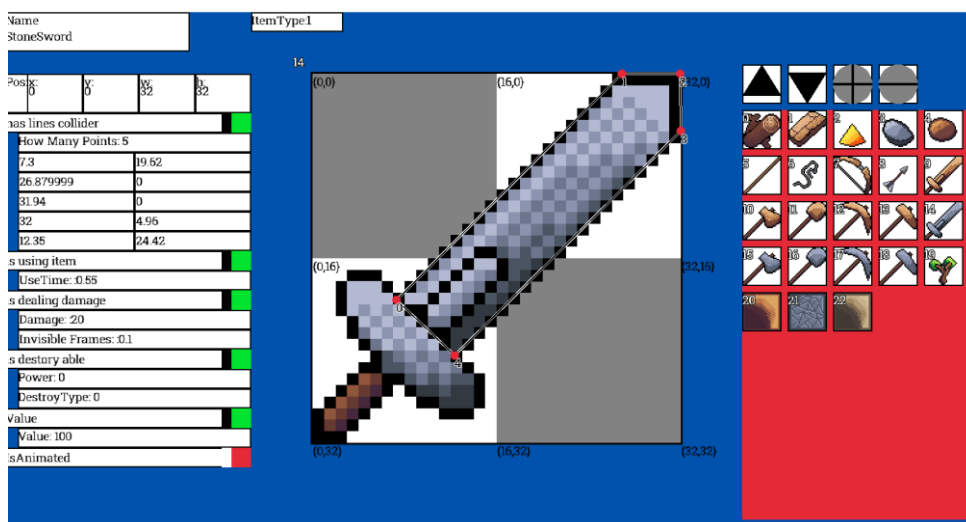
3.5 Zaimplementowane narzędzia pomocnicze

W silniku zostały zaimplementowane 2 dodatkowe narzędzia dla deweloperów, które mogą zostać rozbudowane o dodatkowe funkcjonalności w przyszłości. Pierwsze narzędzie umożliwia dodawanie nowych przedmiotów do rozgrywki. Pozwala ono na ustawienie danych dotyczących obiektów, takich jak wielkość stosu przedmiotów, wartość w sklepie sprzedawców oraz konfigurację colliderów w przypadku broni lub amunicji.

Po lewej stronie narzędzia znajdują się okna pozwalające ustalić, ile takich samych przedmiotów można posiadać w jednym okienku. Obszar centralny odpowiada za prezentację tekstury obiektu, która została przypisana do danego przedmiotu. Natomiast po prawej stronie wyświetlane są wszystkie przedmioty, które aktualnie zostały stworzone. Przyciski strzałki w górę i w dół służą do przewijania listy przedmiotów, a przyciski plus i minus pozwalają na dodawanie lub usuwanie przedmiotów z dostępnych w grze.



Rysunek 25. Okno pozwalające dodawać nowy przedmiot do rozgrywki.



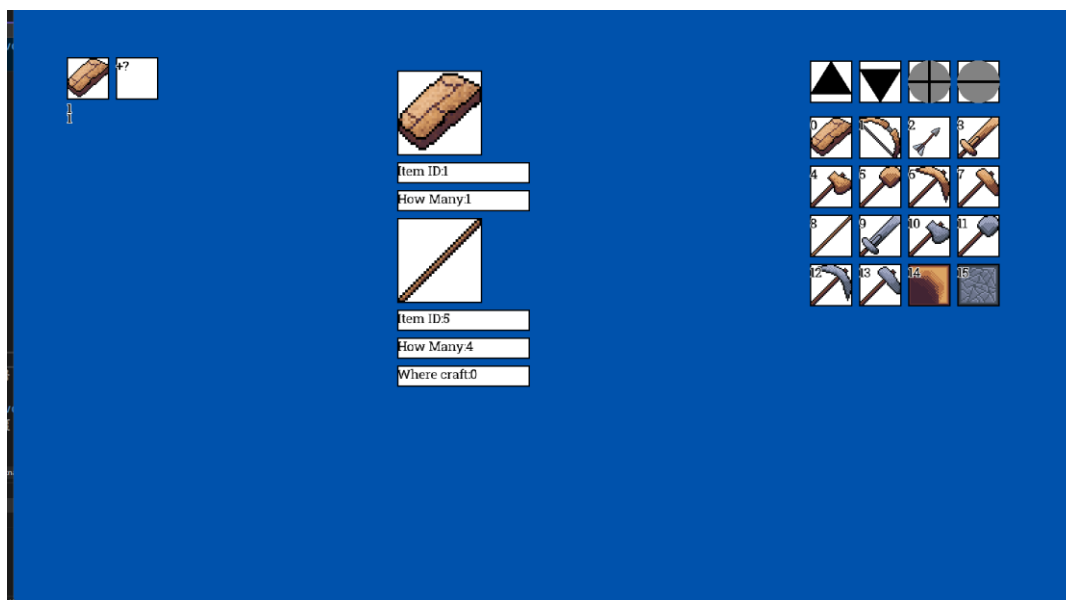
Rysunek 26. Okno pozwalające dodawać nowy przedmiot do rozgrywki przykład z bronią.

Drugim narzędziem jest narzędzie umożliwiające dodawanie nowych receptur, które określają sposób tworzenia danego przedmiotu. Na przykład, możemy ustalić, że aby wytworzyć 4 patyki, potrzebujemy 1 deski. Ta funkcjonalność pozwala na definiowanie zależności pomiędzy różnymi przedmiotami, tworząc w ten sposób reguły, według których mogą być one wytwarzane w grze.

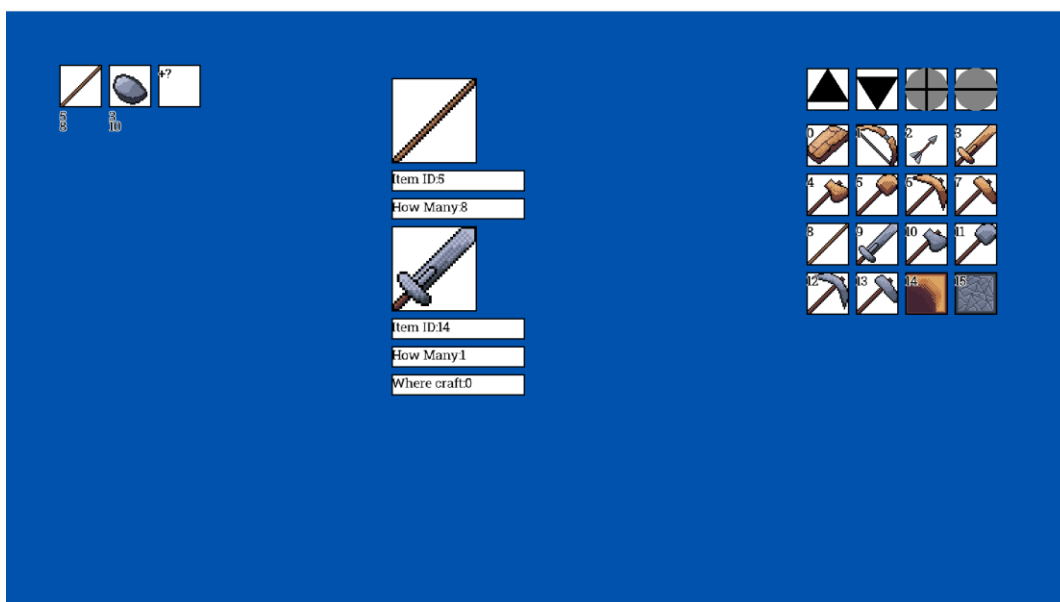
Po lewej stronie narzędzia widoczne są przedmioty, z których składamy przedmiot do stworzenia. Kliknięcie pustego kafelka pozwala na dodanie nowego przedmiotu do receptury. Aby usunąć przedmiot z potrzebnych do stworzenia, możemy kliknąć prawym przyciskiem myszy.

W centralnej części narzędzia pierwsze okienko wizualizuje przedmiot, który dodajemy do receptury. Poniżej znajduje się ID wybranego przedmiotu oraz ilość potrzebna do stworzenia. Czwarte okno reprezentuje przedmiot, który zostanie stworzony z receptury, jego ID oraz ilość powstających przedmiotów. Ostatnie okienko odpowiada za typ wyliczeniowy określający, gdzie można stworzyć przedmiot.

Przechodząc w prawo, znajduje się lista wszystkich receptur, które zostały dodane, analogicznie do tworzenia przedmiotów. Cztery przyciski umożliwiają dodawanie, usuwanie receptur oraz przewijanie menu.



Rysunek 27. Okno z recepturami dostępne w grze



Rysunek 28. Okno z recepturami dostępne w grze przykład z 2 przedmiotami potrzebnymi.

Po wejściu do gry i rozwinięciu ekwipunku, pod głównym oknem ekwipunku znajduje się okienko dotyczące tworzenia przedmiotów. Po kliknięciu klawisza F1 pojawiają się kolejne receptury, do których brakuje nam przedmiotów.



Rysunek 29. Przedstawia ekwipunek gracza oraz okna z przedmiotami do tworzenia.

Przedmioty, które można wytworzyć, są widoczne. Jeśli przedmiot ma zielone tło, oznacza to, że można go stworzyć z posiadanych już przedmiotów. Natomiast czerwone tło wskazuje na brak potrzebnych przedmiotów do stworzenia danego obiektu.



Rysunek 30. Przedstawienie stanu najechania myszą na dany przedmiot do tworzenia

Po najechaniu kursorem na przedmiot, widzimy opis przedmiotu oraz listę składników wykorzystywanych do jego wytworzenia. Jeśli przedmiot wymagany do tworzenia ma czerwone tło, oznacza to, że nie mamy go w ekwipunku. Jeśli jest żółte, brakuje nam odpowiedniej liczby, ale jednocześnie posiadamy co najmniej jedną sztukę w plecaku. Kolor zielony wskazuje, że posiadamy wystarczającą liczbę potrzebnych przedmiotów.



Rysunek 31. Przedstawienie stanu najechania myszą na dany gdy nie posiadamy odpowiedniej ilości przedmiotów.

3.6 Wzorce Projektowe.

Przy implementacji silnika oraz gdy powstało kilka wzorców projektowych, o których warto wspomnieć[13].

3.6.1 Singleton

Singleton[14] to kreacyjny wzorec projektowy, który zapewnia istnienie tylko jednego egzemplarza obiektu w danym czasie. Każdy obiekt może uzyskać dostęp do singletona, ale nie jest możliwe samodzielne tworzenie obiektu klasy singletonowej ani wywoływanie destruktora. Aby uzyskać instancję singletona, tworzy się zazwyczaj metodę statyczną, na przykład `getInstance`, która zwraca obiekt singletona. Singletons są często wykorzystywane przy zarządzaniu połączeniami internetowymi czy bazami danych, ponieważ zapewniają kontrolę nad jednym punktem dostępu do tych zasobów.

W projekcie silnika gier, singletonowe klasy fabryk powstały jako kontenery różnych obiektów. Na przykład, dla wszystkich przedmiotów w grze można utworzyć fabrykę singletonową, która posiada wszystkie przedmioty wczytane z pliku `Items.json`. Dzięki temu jedna instancja fabryki przechowuje informacje o wszystkich przedmiotach i umożliwia dostęp do nich w różnych miejscach w grze bez konieczności tworzenia wielu kopii tych danych.

```
> /// ...
class ItemFactory :Factory
{
    std::vector<Item*> objects;
    static ItemFactory* factory;
    > /// ...
    ItemFactory();
    > /// ...
    ~ItemFactory();
public:
    > /// ...
    static ItemFactory* getFactory();
    > /// ...
    void clearFactory();
    > /// ...
    Item* getObject(int ID);
    > /// ...
    Item* getObject(std::string name);

```

Rysunek 32. Budowa fabryki przedmiotów metody oraz przechowywane dane.

Fabryka przechowuje wektor obiektów, które występują w grze, oraz zawiera zmienną statyczną, która przechowuje odpowiedni obiekt. Metoda `getFactory` ma za zadanie zwrócić obiekt przechowywany w zmiennej `factory`. Jeśli obiekt jeszcze nie istnieje, metoda ta utworzy nowy obiekt i zwróci go. Natomiast metoda `clearFactory` odpowiada za usunięcie fabryki z pamięci. Dzięki metodzie `getObject` i podaniu identyfikatora przedmiotu możemy otrzymać nowy obiekt o dokładnie takich samych właściwościach jak obiekt o danym identyfikatorze.


```

~ItemFactory* ItemFactory::getFactory()
{
    if (factory == NULL)
        factory = new ItemFactory();
    return factory;
}

```

Rysunek 33. Metoda zwracająca obiekt klasy *ItemFactory* jednocześnie tworzy obiekt klasy jeśli jeszcze nie powstał.

```

ItemFactory::ItemFactory()
{
    nlohmann::json j;
    std::ifstream reader;
    reader.open("Items.json");
    if (reader.is_open())
        reader >> j;
    reader.close();
    for (int i = 0; i < j.size(); i++)
    {
        nlohmann::json itemRead = j[i];
        ItemClass type = (ItemClass)itemRead["ItemClass"];
        switch (type)
        {
            case ItemClass::StackItem:
                objects.push_back(new StackItem(itemRead));
                break;
            case ItemClass::ToolItem:
                objects.push_back(new ToolItem(itemRead));
                break;
            case ItemClass::Bow:
                objects.push_back(new Bow(itemRead));
                break;
        }
    }
}

```

Rysunek 34. Konstruktor wczytujący obiekty z pliku *Json*

3.6.2 Prototyp

Prototyp jest kolejnym wzorcem projektowym kreacyjnym, który jest używany w projekcie w połączeniu ze wzorcem singleton. W kontekście przykładu z fabryką przedmiotów, gdzie wszystkie obiekty są przechowywane w jednym wektorze, po podaniu odpowiedniego identyfikatora obiektu otrzymujemy dokładną kopię tego obiektu. Metoda klonowania jest metodą wirtualną, co oznacza, że jeśli klasa *StackItem* dziedziczy po klasie *Item*, próba skopiowania obiektu zwróci obiekt klasy *StackItem*.

```

~Item* ItemFactory::getObject(int i)
{
    if (i > -1 && i < objects.size())
        return objects[i]->clone();
    return NULL;
}

```

Rysunek 35. Zwraca sklonowany obiekt jeśli zostało podane odpowiednie *i* które mieści się w zakresie wektora.


```
virtual Item* clone() { return new Item(*this); }
```

Rysunek 36. Metoda kopiująca dane z klasy Item.

```
#pragma once
#include "Item.h"
class StackItem :
public Item
{
```

Rysunek 37. Klasa StackItem dziedziczy po klasie Item co pozwala sklonować odpowiedni item przy wywołaniu metody clone

```
virtual StackItem* clone() { return new StackItem(*this); }
```

Rysunek 38. Przysłonięta metoda clone kopiująca dane z klasy StackItem

```
StackItem::StackItem(StackItem& item):Item(item)
{
    stackMaxSize = item.stackMaxSize;
    stackSize = item.stackSize;
    sprite = new SpriteController(*item.sprite);
}
```

Rysunek 39. Konstruktor kopiujący dane z obiektu o tej samej klasie

Aby system działał poprawnie, każdy obiekt musi posiadać konstruktor umożliwiający kopiowanie danych na podstawie innego obiektu, z którego są one kopiowane. Przykładowo, w przypadku klasy StackItem można zauważyć, że są kopiowane maksymalna wielkość przechowywanych przedmiotów, obecna ilość oraz tekstura.

4. Testy wydajnościowe

Ten rozdział będzie oceniał wydajność wykonanych rozwiązań oraz jak wpływają one na pracę systemu, porównując działanie systemu z zastosowanymi rozwiązaniami i bez nich, przy wykorzystaniu programu RivaTuner[12].

4.1 Test drzewa czwórkowego

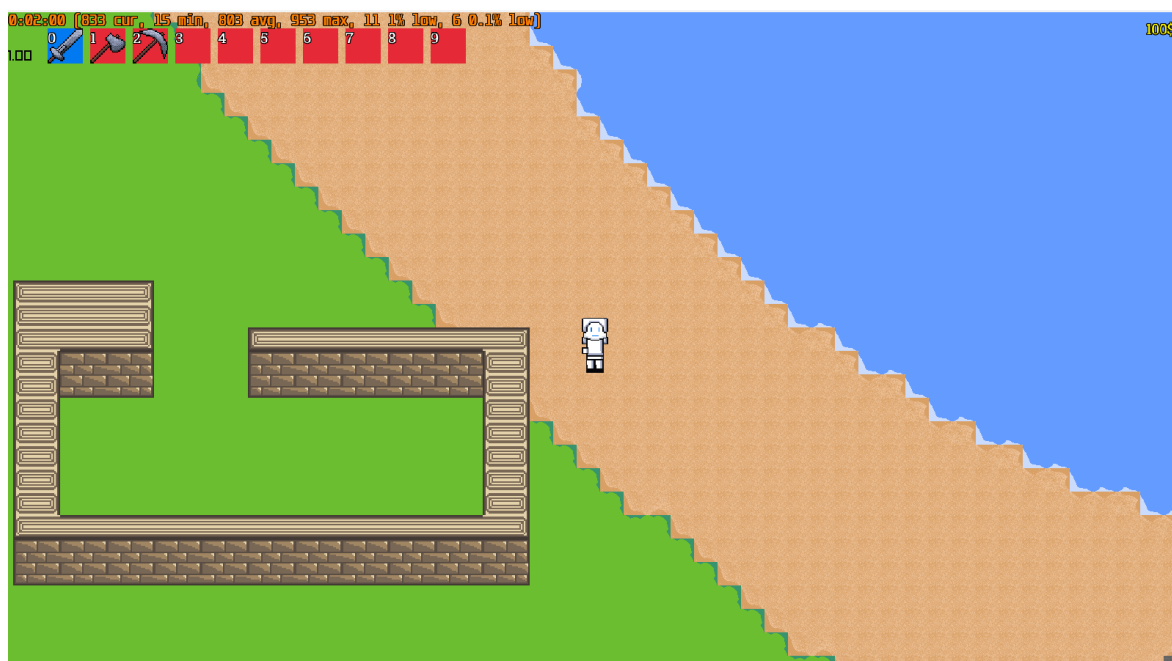
Aby ocenić wydajność drzewa czwórkowego w systemie, przeprowadzono 2 próby testowe. Pierwsza próba polegała na standardowym wykorzystaniu drzewa czwórkowego, podczas gdy drugi test opierał się na zmianie wartości otwierania się poszczególnych ćwiartek drzewa na dużą liczbę obiektów. W pierwszym teście drzewo dzieliło się na cztery, gdy było więcej niż 8 obiektów. Następnie przeprowadzono test, w którym drzewo musiało obsłużyć 10000 obiektów.

Test polegał na 2-minutowej rozgrywce, w której gracz niszczył kilka drzew, a następnie budował pomieszczenie i umieszczał w nim 100 obiektów zombie. Następnie pomieszczenie zostało otwarte, a wszystkie zombi zostały zabite. Następnie dokonano analizy wyników.

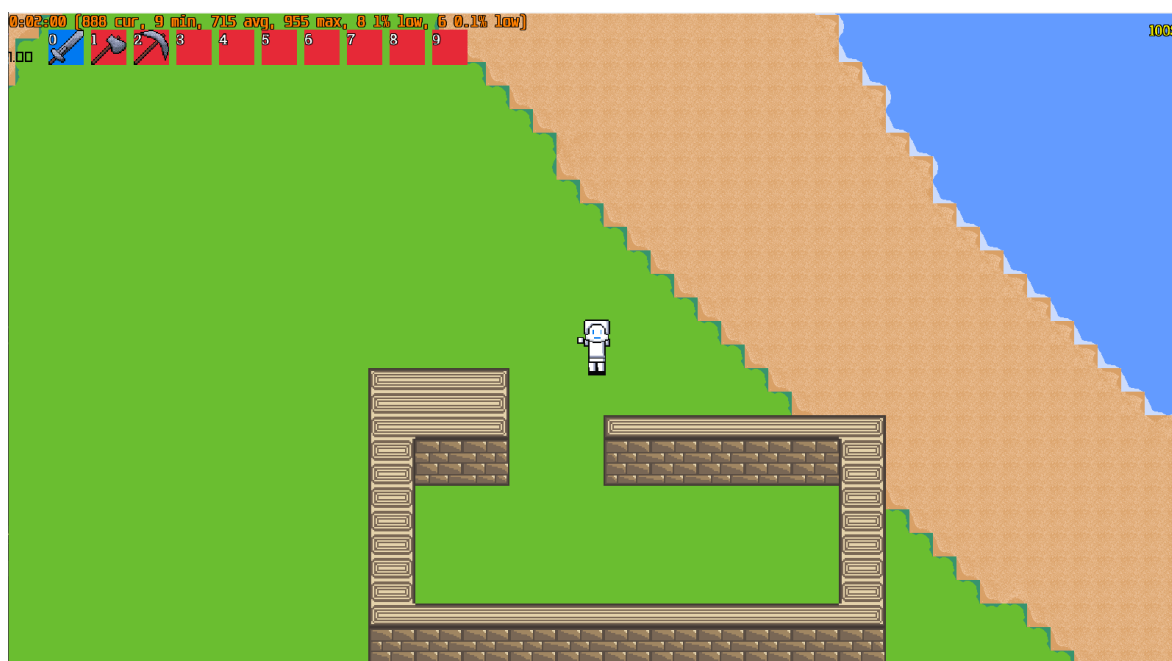
```
if (IsKeyDown(KEY_LEFT_SHIFT))
{
    if (IsMouseButtonPressed(MOUSE_BUTTON_MIDDLE))
    {
        for (int i = 0; i < 100; i++)
        {
            GameObject* o = EnemyFactory::getFactory()->getObject(0);
            if (o)
            {
                o->setMovePos({ cursorPos.x+i%100,cursorPos.y+i/100 });
                addObject(o);
            }
        }
    }
}
```

Rysunek 40. Funkcjonalności w kodzie pozwalająca stworzyć 100 zombi

Dla dokładności pomiaru dotyczącego dodawanych obiektów, zostało przypisane tworzenie 100 zombi pod przyciski lewy Shift oraz środkowy przycisk myszy.



Rysunek 41. Próba gdzie drzewo czwórkowe działało normalnie



Rysunek 42. Próba gdzie drzewo czwórkowe było ograniczone w działaniu.

Wyniki testów wygląda następująco

	Średnie FPSy	Minimalne FPSy	Maksymalne FPSy
Normalnie działające drzewo	803	15	953
Drzewo działające od 10000 obiektów	715	9	955

Tabela 1. Klatki na sekundę przy optymalizacji drzewa czwórkowego

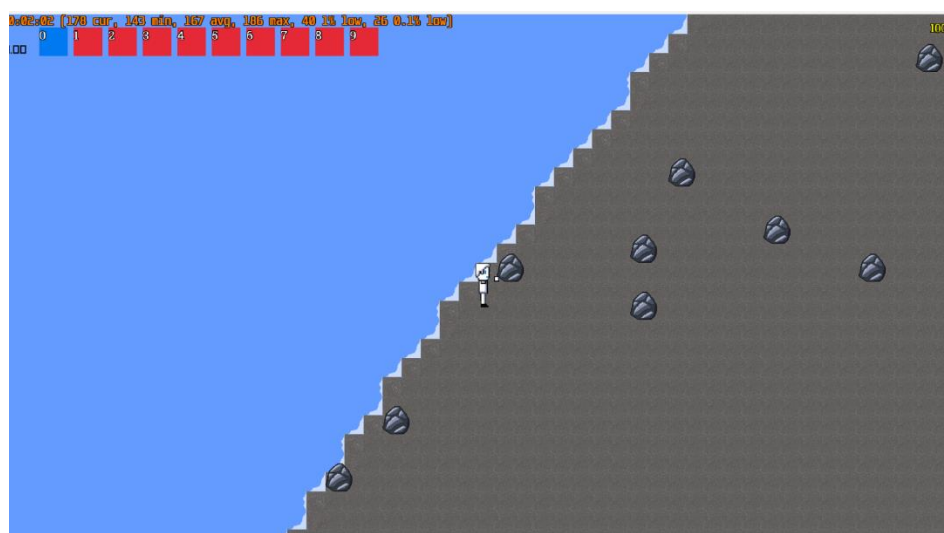
Największa widoczna różnica występuje w średniej liczbie klatek, która wynosi 88 klatek na sekundę. Dane dotyczące minimalnej liczby klatek różnią się zaledwie o 6 klatek. Najniższa liczba klatek wystąpiła podczas zabijania wszystkich 100 obiektów, co jest akceptowalne, gdyż niszczenie tak dużej liczby obiektów podczas ataku zombi na gracza powoduje częste odwołania do dostarczenia danych z małego obszaru.

Natomiast w przypadku maksymalnej liczby klatek różnica jest minimalna - zaledwie 2 klatki na korzyść drzewa działającego w przypadku 10000 obiektów. Dlatego można pominąć ten pomiar, ponieważ dla 900 klatek różnica 2 klatek nie ma znaczącego wpływu, zwłaszcza w porównaniu do 10 klatek na sekundę.

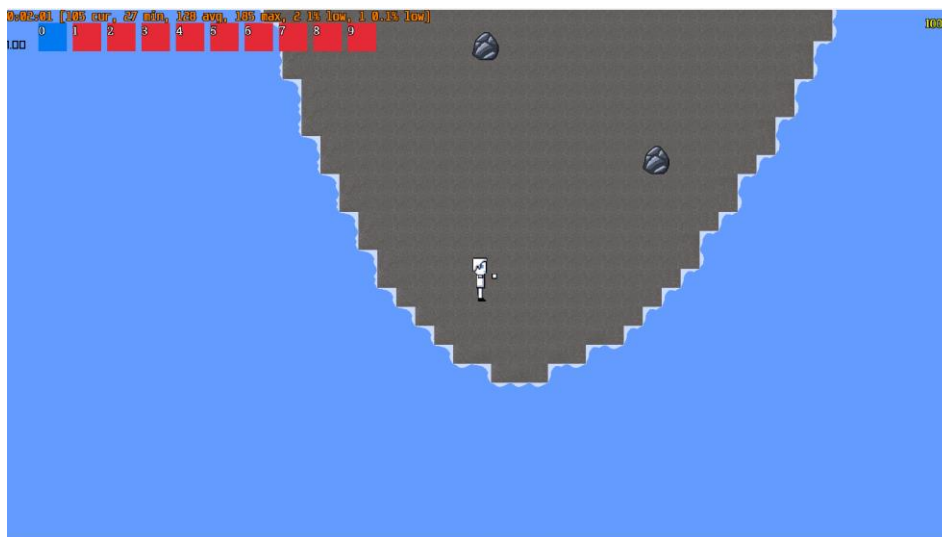
Można stwierdzić, że drzewo czwórkowe pozytywnie wpływa na płynność rozgrywki. Zarówno minimalne, jak i średnie liczby klatek jednoznacznie pokazują, że system działa poprawnie i optymalizuje działanie całego projektu.

4.2 Test zapisu na kilku plikach

W tym teście zostanie przeprowadzony pomiar liczby klatek na sekundę poprzez poruszanie się przez 2 minuty w prawo i analizę liczby klatek na sekundę. W pierwszym teście obszar o wymiarach 10x10 będzie zapisywany do pliku, natomiast w drugim testowane będą obszary o rozmiarze 100 000x100 000. Test ma na celu sprawdzenie, jaki wpływ na wczytywanie mapy ma wielkość danych. By można test wykonać poprawnie zostanie zwiększony obszar renderowania terenu oraz gracz będzie przemieszczał się 5 razy szybciej by wygenerować jak największy obszar



Rysunek 43. Bieg przy zapisywaniu na mniejsze pliki 10x10



Rysunek 44. Bieg przy zapisywaniu na większe pliki 10000x10000

Wyniki testów wygląda następująco

	Średnie FPSy	Minimalne FPSy	Maksymalne FPSy
Pliki mniejsze	167	143	187
Pliki większe	128	27	185

Tabela 2. Klatki na sekundę przy optymalizacji zapisu plików

Przy mniejszych plikach średnia liczba klatek, którą uzyskujemy, wynosiła 39, natomiast minimalna liczba klatek różniła się o 116. Jest to diametralna różnica. Porównując minimalną ilość klatek do średniej wartości pomiaru dla mniejszych plików, różnica wynosi zaledwie 24 klatki na sekundę. Natomiast w przypadku większych plików różnica ta wzrasta do 101 klatek na sekundę, co stanowi wynik prawie 5-krotnie mniejszy niż średnia liczba. Podobnie jak przy poprzednim teście, maksymalna liczba klatek nie dostarcza istotnej informacji.

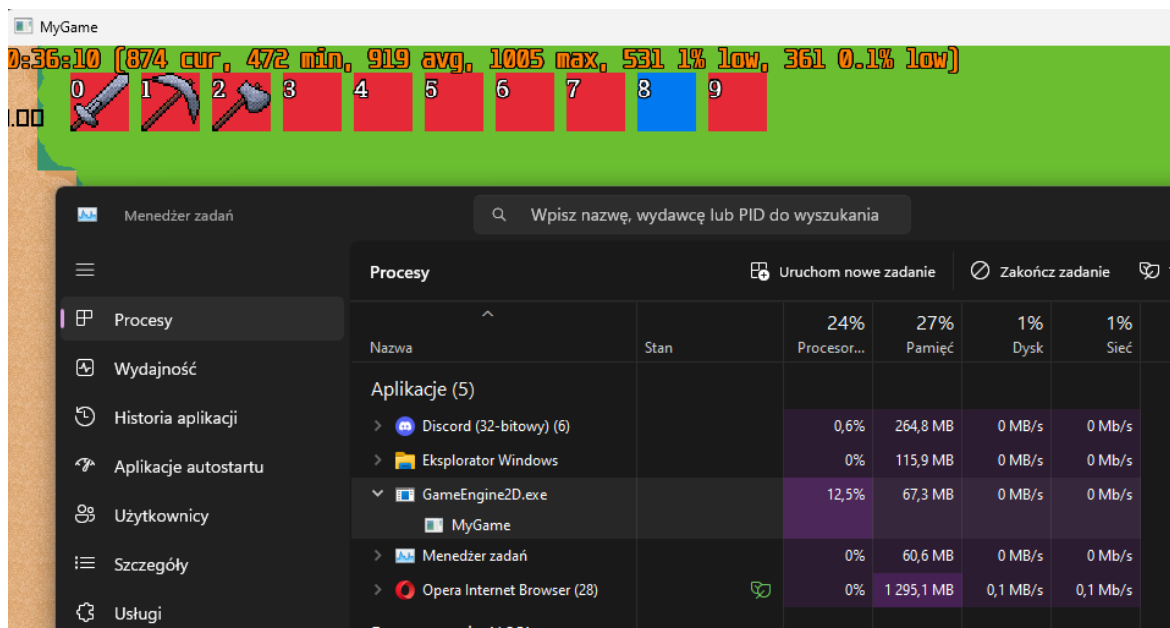
Test wykazał, że zastosowane mechanizmy optymalizacyjne rzeczywiście poprawiają wydajność programu poprzez redukcję wielkości plików.

4.3 Test wydajnościowy

Test wydajnościowy zostanie wykonany na podstawie korzystania z programu przez dłuższy fragment czasu. Po upływie 36 min wynik silnika wyniósł:

- Średnie klatki na sekundę: 919
- Minimalne klatki na sekundę 472
- Maksymalne klatki na sekundę 1005

- Program zajmował 67 MB pamięci RAM



Rysunek 45. Wyniki testowania programu przez 36 min

W trakcie testu przeprowadzano różne aktywności, takie jak eksploracja wygenerowanego terenu, walka z przeciwnikami, budowanie i niszczenie obiektów na mapie. Minimalna liczba klatek na sekundę nie spadła poniżej 472, co stanowi satysfakcjonujący wynik, biorąc pod uwagę, że standardem w branży gier komputerowych jest osiągnięcie 60 klatek na sekundę.

Co więcej, zużycie pamięci RAM było minimalne w porównaniu z innymi aplikacjami działającymi równocześnie. Na przykład, aplikacja Discord zajmowała 264 MB pamięci RAM, podczas gdy przeglądarka internetowa Opera używała 1295 MB.

5. Rozwój silnika

Silnik, który został stworzony, oferuje wiele różnych dróg rozwoju. Jednym z nich jest rozwój generatora świata, który mógłby tworzyć losowe struktury, w których gracze natrafiliby na przeciwników i odkrywaliby nowe przedmioty. Dodanie nowych mechanik, takich jak system głodu, może wprowadzić kolejne aspekty, na które gracze musieliby zwracać uwagę podczas rozgrywki. Rozważa się także opcję umożliwienia uprawy roślin oraz hodowli zwierząt, co mogłoby zapewnić graczom dodatkowe cele związane z przetrwaniem oraz budowaniem własnych osad.

Innym kierunkiem rozwoju jest dodanie trybu multiplayer, który umożliwiłby wielu graczom wspólne eksplorowanie świata gry. Istnieje także możliwość stworzenia dodatkowych narzędzi do konstruowania przedmiotów lub receptur związanych z różnymi aspektami terenu w grze.

Możliwe jest także stworzenie platformy, na której gracze oraz moderatorzy mogliby dodawać kolejne elementy, w sposób podobny do takich gier jak Terraria, Minecraft czy Fortnite. Twórcy modyfikacji gry nadal wprowadzają nowe możliwości, co pozwala graczom na rozszerzanie zawartości gry poprzez dodawanie nowych elementów za pomocą różnorodnych modyfikacji.

Dodatkowo istnieje możliwość ulepszenia gry pod względem graficznym poprzez poprawę tekstur obecnych w grze lub wprowadzenie różnorodnych efektów graficznych za pomocą shaderów. Te efekty mogą symulować światło, co umożliwiłoby stworzenie cyklu dnia i nocy, dodając w ten sposób realizmu do świata gry. Dodanie ścieżki dźwiękowej również może znacząco przyczynić się do zanurzenia gracza w grze, podnosząc poziom zaangażowania i atmosfery.

W przyszłości rozważana może być także ekspansja gry na inne platformy, takie jak Nintendo Switch, PlayStation 5 czy Xbox Series X.

6. Podsumowanie

Wynikiem mojej pracy jest gra stworzona przy użyciu autorskiego silnika graficznego, opartego wyłącznie na bibliotece graficznej. Produkt ten zawiera mechanizmy umożliwiające dodawanie nowych przedmiotów i receptur, które są dostępne dla graczy. Główne cele projektowe zostały zrealizowane, co sprawia, że produkt ten ma wiele możliwości rozwoju, pozwalając mu na ewentualne ukończenie i wydanie na platformach gamingowych, takich jak np. platforma Steam.

Dodatkowo produkt został zoptymalizowany w taki sposób, aby komfort korzystania z niego był pozytywny dla użytkowników. Przeprowadzone zostały testy wydajnościowe, które pokazują różnice osiągnięte dzięki zastosowaniu odpowiednich struktur i rozwiązań optymalizacyjnych.

Bibliografia

- [1] Bestsellery od wt., 26 grudnia 2023 do wt., 2 stycznia 2024 Polska Steam
[Data ostatniego odwiedzenia: 04.01.2024]
<https://store.steampowered.com/charts/topsellers/PL/2023-12-26>
- [2] Artykuł na temat podejścia The Legend of Zelda: Breath of The Wild do otwartości świata
[Data ostatniego odwiedzenia: 13.01.2024]
<https://www.ppe.pl/publicystyka/319986/od-tla-do-bohatera-czyli-jak-zelda-zmienila-podejscie-do-otwartego-swiata-w-grach-wideo.html>
- [3] Artykuł o środkach transportu w świecie ArcheAge
[Data ostatniego odwiedzenia: 13.01.2024]
<https://www.gry-online.pl/S024.asp?ID=1554&PART=60>
- [4] Film opowiadający o systemie drzewa czwórkowego oraz implementacja w c++
[Data ostatniego odwiedzenia: 10.01.2024]
https://www.youtube.com/watch?v=ASAowY6yJII&ab_channel=javidx9
- [5] Artykuł o działaniu algorytmu szukania ścieżki A*
[Data ostatniego odwiedzenia: 10.01.2024]
<https://happycoding.io/tutorials/libgdx/pathfinding>
- [6] Książka: Język C++. Szkoła programowania Autora [Stephen Prata](#)
- [7] Strona biblioteki graficznej z przykładami
[Data ostatniego odwiedzenia: 14.01.2024]
<https://www.raylib.com>
- [8] Środowisko programistyczne
[Data ostatniego odwiedzenia: 14.01.2024]
<https://visualstudio.microsoft.com/pl/>
- [9] Użyta biblioteka do korzystania z plików Json
[Data ostatniego odwiedzenia: 14.01.2024]
<https://github.com/nlohmann/json>
- [10] Biblioteka szumów do generowania terenu.
[Data ostatniego odwiedzenia: 14.01.2024]
<https://github.com/Auburn/FastNoiseLite?tab=readme-ov-file>
- [11] Wykład tłumaczący sposób generowania terenu w minecraft
[Data ostatniego odwiedzenia: 09.01.2024]
<https://youtu.be/ob3VwY4JyzE>
- [12] Program do pomiaru klatek na sekundę
[Data ostatniego odwiedzenia: 05.01.2024]
<https://www.guru3d.com/download/rtss-rivatuner-statistics-server-download/>
- [13] Książka Dive into Design Patterns autora Alexander Shvets
- [14] Artykuł opowiadający o wzorcu projektowym Singleton
[Data ostatniego odwiedzenia: 14.01.2024]
<https://refactoring.guru/design-patterns/singleton>
- [15] Artykuł opowiadający o wzorcu projektowym Prototype
[Data ostatniego odwiedzenia: 14.01.2024]
<https://refactoring.guru/design-patterns/prototype>

Spis ilustracji

Rysunek 1. Bestsellery w tygodniu 100 najlepszych produktów według przychodu z platformy Steam od wt., 26 grudnia 2023 do wt., 2 stycznia 2024	3
Rysunek 2. Drzewo czwórkowe wizualizacja struktury	8
Rysunek 3. Rysunek przedstawia katalog z zapisanymi regionami odpowiednio ponumerowanymi.	9
Rysunek 4. Algorytm A* szukający ścieżki z białego kwadratu do żółtego punktu końcowego	10
Rysunek 5. Struktura drzewa w kodzie programu	13
Rysunek 6. Metody drzewa czwórkowego	14
Rysunek 7. Prywatne metody klasy pozwalające zwiększyć drzewo oraz ucięcie fragmentu drzewa	15
Rysunek 8. Przedstawienie struktury drzewa gdy w jego fragmencie znajduje się 8 obiektów.	15
Rysunek 9. Przedstawia strukturę drzewa po dodaniu dziewiątego obiektu	16
Rysunek 10. Przedstawienie struktury drzewa gdy zostanie dodane więcej obiektów w danej ćwiartce.	16
Rysunek 11. Zapisywanie danych abstrakcyjnej klasy GameObject.....	17
Rysunek 12. Wczytywanie danych abstrakcyjnej klasy GameObject	17
Rysunek 13. Przeciwnik wyszukuje drogi do gracza.	18
Rysunek 14. Przeciwnik przeszedł przez wybudowaną przeszkodę.....	19
Rysunek 15. Pierwszy system szukania ścieżki	20
Rysunek 16. Szum odpowiedzialny za konstrukcje na mapie.	21
Rysunek 17. Szum odpowiedzialny za określenie występowania wody.	21
Rysunek 18. Szum odpowiedzialny za określenie przyjazności terenu.	22
Rysunek 19. Szum odpowiedzialny za określenie temperatury występującej na danym obszarze	22
Rysunek 20. Fragment kodu determinujący teren jaki zostanie wygenerowany	23
Rysunek 21. Wygenerowany teren zimowy	23
Rysunek 22. Wygenerowana pustynia.....	24
Rysunek 23. Wygenerowany Las z plażą	24
Rysunek 24. Przykładowa wygenerowana wyspa	24
Rysunek 25. Okno pozwalające dodawać nowy przedmiot do rozgrywki.	25
Rysunek 26. Okno pozwalające dodawać nowy przedmiot do rozgrywki przykład z bronią.....	25
Rysunek 27. Okno z recepturami dostępne w grze.....	26
Rysunek 28. Okno z recepturami dostępne w grze przykład z 2 przedmiotami potrzebnymi.	27
Rysunek 29. Przedstawia ekwipunek gracza oraz okna z przedmiotami do tworzenia.	27
Rysunek 30. Przedstawienie stanu najechania myszą na dany przedmiot do tworzenia	28
Rysunek 31. Przedstawienie stanu najechania myszą na dany gdy nie posiadamy odpowiedniej ilości przedmiotów.....	28
Rysunek 32. Budowa fabryki przedmiotów metody oraz przechowywane dane.	29
Rysunek 33. Metoda zwracająca obiekt klasy ItemFactory jednocześnie tworzy obiekt klasy jeśli jeszcze nie powstała.....	30
Rysunek 34. Konstruktor wczytujący obiekty z pliku Json	30
Rysunek 35. Zwraca sklonowany obiekt jeśli zostało podane odpowiednie i które miesi się w zakresie wektora.	30
Rysunek 36. Metoda kopiująca dane z klasy Item.....	31
Rysunek 37. Klasa StackItem dziedziczy po klasie Item co pozwala sklonować odpowiedni item przy wywołaniu metody clone.....	31
Rysunek 38. Przysłonięta metoda clone kopiująca dane z klasy StackItem	31
Rysunek 39. Konstruktor kopiujące dane z obiektu o tej samej klasie.....	31
Rysunek 40. Funkcjonalności w kodzie pozwalająca stworzyć 100 zombi.....	32
Rysunek 41. Próba gdzie drzewo czwórkowe działało normalnie	33
Rysunek 42. Próba gdzie drzewo czwórkowe było ograniczone w działaniu.	33
Rysunek 43. Bieg przy zapisywaniu na mniejsze pliki 10x10.....	34
Rysunek 44. Bieg przy zapisywaniu na większe pliki 10000x10000	35
Rysunek 45. Wyniki testowania programu przez 36 min	36

Spis Tabel

Tabela 1. Klatki na sekundę przy optymalizacji drzewa czwórkowego	33
Tabela 2. Klatki na sekundę przy optymalizacji zapisu plików	35